

Práctica Exploratoria: Implementación de una aplicación Java y una API REST Serverless en AWS

Alumno/a: Marc Campos Roca

Alumno/a: Arnau Colominas Illa

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

Facultat d'Informàtica de Barcelona



Contenido

Introducción	1
Hashicorp Terraform	1
Amazon Web Services	1
Serverless	1
Decisiones de diseño	2
Esquema del diseño	3
Frontend	4
AWS Certificate Manager (ACM)	4
Aplicacion Java Cliente	4
AWS Elastic Beanstalk	5
Amazon Route53	8
Backend	9
Amazon API Gateway	9
AWS Lambda	13
Amazon Cognito	17
Amazon DynamoDB	20
Amazon S3	21
Escalabilidad del diseño	22
Repositorios de código consultados	22
Bibliografía consultada	22

Introducción

En este informe se detallan las decisiones que hemos tomado durante la implementación de la Práctica Exploratoria: Implementación de una aplicación Java y una API REST Serverless en AWS.

Hashicorp Terraform

Para crear toda la infraestructura para este proyecto lo hemos definido usando Terraform. Terraform es un programa que nos permite gestionar toda la infraestructura desplegada como Infrastructure as Code (IaC). De este manera tenemos definido con código toda nuestra infraestructura y podemos saber en todo momento lo que tenemos desplegado, así como una mejor gestión del cambio, generar estándares o bien implementar una política de CI/CD (Continuous Integration/Continuous Delivery) subiendo el código a un repositorio Git y uniendolo con pipelines.

Terraform puede conectar con múltiples proveedores de Cloud públicos como Amazon Web Services (AWS), Google Cloud Platform (GCP), Microsoft Azure, etc..

El código se encuentra en el directorio **Infra_AWS**

Amazon Web Services

Hemos decidido desplegar la aplicación en el Cloud de Amazon ya que tiene mucha versatilidad y nos permite usar servicios serverless (sin preocuparnos de la gestión de la infraestructura subyacente).

Amazon Web Services (AWS abreviado) es una colección de servicios de computación en la nube pública (también llamados servicios web) que en conjunto forman una plataforma de computación en la nube, ofrecidas a través de Internet por Amazon.com. Es usado en aplicaciones populares como Dropbox, Foursquare, HootSuite. Es una de las ofertas internacionales más importantes de la computación en la nube y compite directamente contra servicios como Microsoft Azure, Google Cloud Platform y IBM Cloud. Es considerado como un pionero en este campo.

Serverless

En general, el término *serverless* se emplea para referirse al modelo de computación según el cual el proveedor de la capa de computación nos permite ejecutar durante un periodo de tiempo determinado porciones de código denominadas "funciones" sin necesidad de hacernos cargo de la infraestructura subyacente que se provisiona para dar el servicio. En este modelo, el proveedor se encarga de ofrecer los recursos de forma transparente, de escalarlos automáticamente si crece la demanda y de liberarlos cuando no son utilizados, definiendo una serie de restricciones referentes al procesamiento y un modelo de pago por el consumo de los recursos derivados de la ejecución

Decisiones de diseño

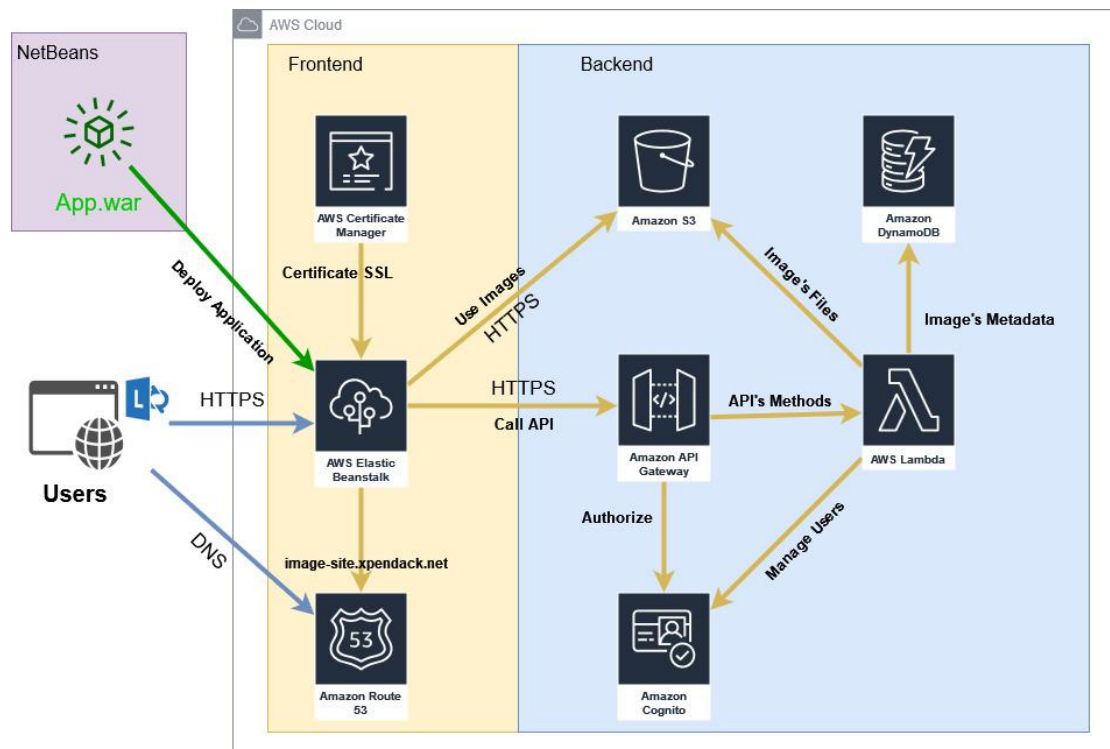
Para todo el diseño del proyecto hemos intentado usar los máximos servicios Serverless. Todo el backend está realizado con servicios Serverless.

Para el frontend de la aplicación Java hemos usado AWS Elastic Beanstalk, que aunque es cierto que no es serverless (la aplicación Java se despliega en una instancia EC2) nos da una capa de abstracción que nos facilita su despliegue y gestión.

Servicios AWS usados:

- Amazon Route53
 - Para la gestión del dominio y los registros DNS
- AWS Certificate Manager
 - Para la obtención del certificado SSL para el Frontend de la aplicación.
- AWS Elastic Beanstalk
 - Para el despliegue de la aplicación Java
- Amazon API Gateway
 - Para la creación de la API REST
- AWS Lambda
 - Para el desarrollo de los métodos de la API REST. Escritos en Python 3.9.
- Amazon S3
 - Para el almacenamiento de las imágenes físicas y su posterior publicación a Internet
- Amazon DynamoDB
 - Base de datos NoSQL usada para el almacenamiento de los metadatos de las imágenes.
- Amazon Cognito
 - Para la gestión de usuarios y la autorización de la API REST.

Esquema del diseño



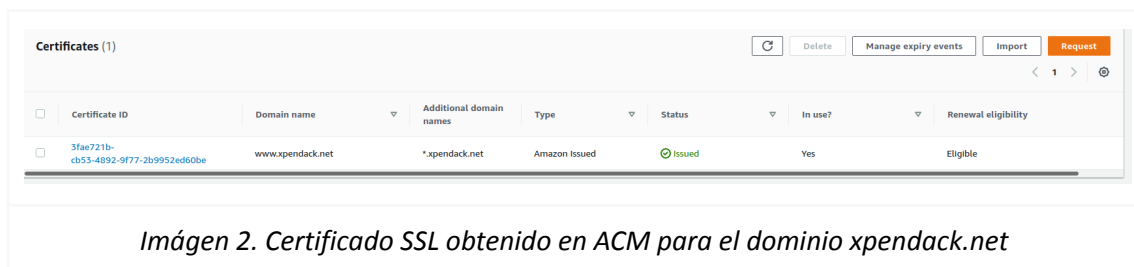
Imágen 1. Arquitectura del sistema y los servicios AWS

Frontend

AWS Certificate Manager (ACM)

AWS Certificate Manager es un servicio que nos permite aprovisionar, administrar e implementar con facilidad certificados de capa de conexión segura/seguridad de la capa de transporte (SSL/TLS) públicos y privados para su uso con servicios de AWS y recursos internos conectados. Los certificados de SSL/TLS se usan para proteger comunicaciones por red y para definir la identidad de sitios web mediante Internet y recursos en redes privadas. AWS Certificate Manager elimina el arduo proceso manual de compra, carga y renovación de los certificados de SSL/TLS.

Hemos usado un certificado SSL para el dominio *.xpendack.net y lo hemos añadido a la configuración del AWS Elastic Beanstalk.



Aplicacion Java Cliente

Hemos usado la aplicación Java cliente que hemos realizado en la Práctica 4. Hemos adaptado el código para que se integre con la nueva API desplegada con Amazon API Gateway.

Para poder desarrollar la parte del cliente Java en Netbeans hemos usado el mismo jdk usado por AWS Elastic Beanstalk descargandolo y añadiendolo a NetBeans desde:

<https://docs.aws.amazon.com/corretto/latest/corretto-8-ug/downloads-list.html>

El servidor hemos escogido Glassfish Server 5.1

Algunos cambios a destacar:

- Las imágenes no se descargan, se muestran a través de S3. Cada objeto Image dispone de un atributo con la URL de la imagen.

Después de realizar el build de la aplicación obtenemos un WAR en la ruta APP_Folder/target/ y luego lo subimos a AWS Elastic Beanstalk para que se despliegue en las instancias EC2.

El código de la aplicación se encuentra en el directorio **Application_Webclient**.

AWS Elastic Beanstalk

AWS Elastic Beanstalk es un servicio fácil de utilizar para implementar y escalar servicios y aplicaciones web desarrollados con Java, .NET, PHP, Node.js, Python, Ruby, Go y Docker en servidores familiares como Apache, Tomcat, Nginx, Passenger e IIS.

Solo tenemos que cargar nuestra aplicación y Elastic Beanstalk administra de manera automática la implementación, desde el aprovisionamiento de la capacidad, el equilibrio de carga y el escalado automático hasta la monitorización del estado de la aplicación. Al mismo tiempo, tendremos el control absoluto de los recursos de AWS que alimentan nuestra aplicación y podremos acceder a los recursos subyacentes cuando queramos.

Hemos escogido el motor "64bit Amazon Linux 2 v4.2.8 running Tomcat 8.5 Corretto". Es decir, un servidor Tomcat 8.5 corriendo sobre instancias con Amazon linux 2 como sistema operativo.

Las instancias son del tipo t3.medium, 2 CPU, 4 GB RAM y hasta 5 Gbps. Si viéramos que la performance es pobre podríamos subir el tipo de instancia.

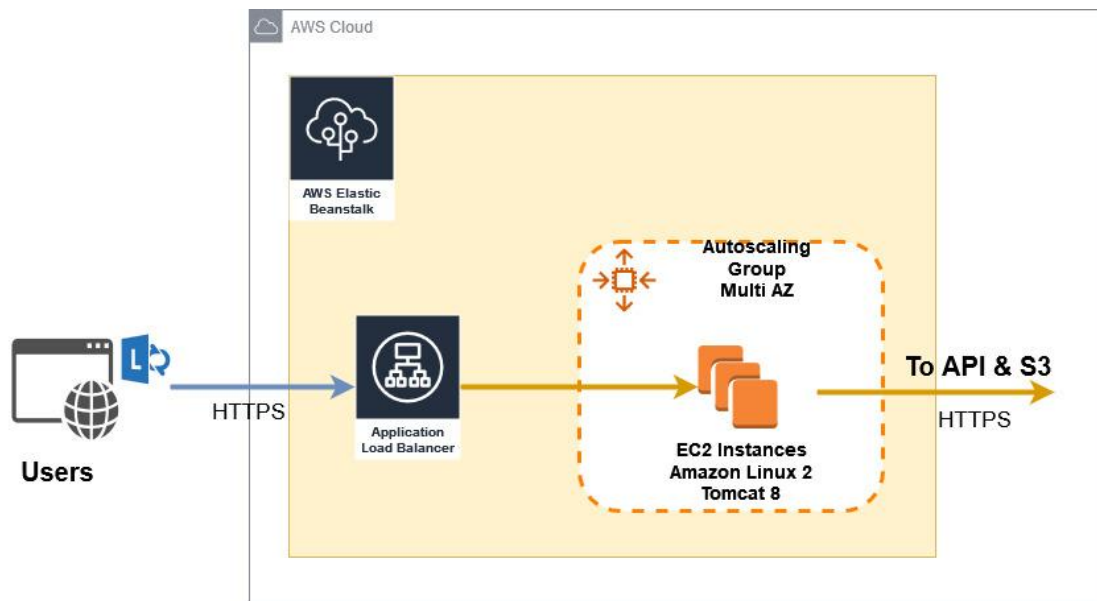
Nos genera un Application Load Balancer y un grupo de instancias EC2 con un grupo de Autoescalado que siempre dispondrá de como mínimo dos instancias levantadas. En caso de pérdida de servicio de una instancia (debido a un problema interno), la otra dará servicio y automáticamente se levantará una nueva para sustituir a la problemática.

Además hemos configurado unos threshold para añadir o quitar instancias en nuestro grupo de instancias.

- Si durante 5 minutos el promedio de consumo de CPU de todas las instancias es superior a 70% se añadirá una instancia al grupo.
- Si durante 5 minutos el promedio de consumo de CPU de todas las instancias es inferior a 30% se eliminará una instancia del grupo.

En el Application Load Balancer se añade el certificado SSL obtenido en ACM. Y se habilita sticky sessions con una duración de un día. Es decir, como no realizamos gestión de sesiones, el balanceador guardará durante un día las sesiones para que el mismo cliente acceda a la misma instancia EC2.

Las instancias EC2 se levantan en multi-AZ (Availability Zone) para proporcionarnos disponibilidad en caso de caída de una zona de disponibilidad.



Imágen 3. Arquitectura interna AWS Elastic Beanstalk

El dashboard de la aplicación nos muestra el estado de la misma y podemos consultar los logs para ver posibles problemas.

Elastic Beanstalk > Environments > image-manager-pro

image-manager-pro
image-manager-pro-eba-dguu7ap9.eu-west-1.elasticbeanstalk.com [c-e55ap44bb]
Application name: ImageManager

Refresh Actions

Health
Ok
Causes

Running version
Upload and deploy

Platform
Tomcat 8.5 with Corretto 8 running on 64bit Amazon Linux 2/4.2.8
Change

Recent events Show all

Time	Type	Details
2021-12-12 17:07:50 UTC+0100	INFO	Environment health has transitioned from Pending to Ok. Initialization completed 43 seconds ago and took 3 minutes.
2021-12-12 17:07:28 UTC+0100	INFO	Successfully launched environment: image-manager-pro
2021-12-12 17:07:27 UTC+0100	INFO	Application available at image-manager-pro-eba-dguu7ap9.eu-west-1.elasticbeanstalk.com.
2021-12-12 17:06:57 UTC+0100	INFO	Instance deployment completed successfully.
2021-12-12 17:06:51 UTC+0100	INFO	Created Load Balancer listener named: arn:aws:elasticloadbalancing:eu-west-1:004914726163:listener/app/awseb-12EDL8NEMF9JN/3badf06228cc15dd/35561a376718f138

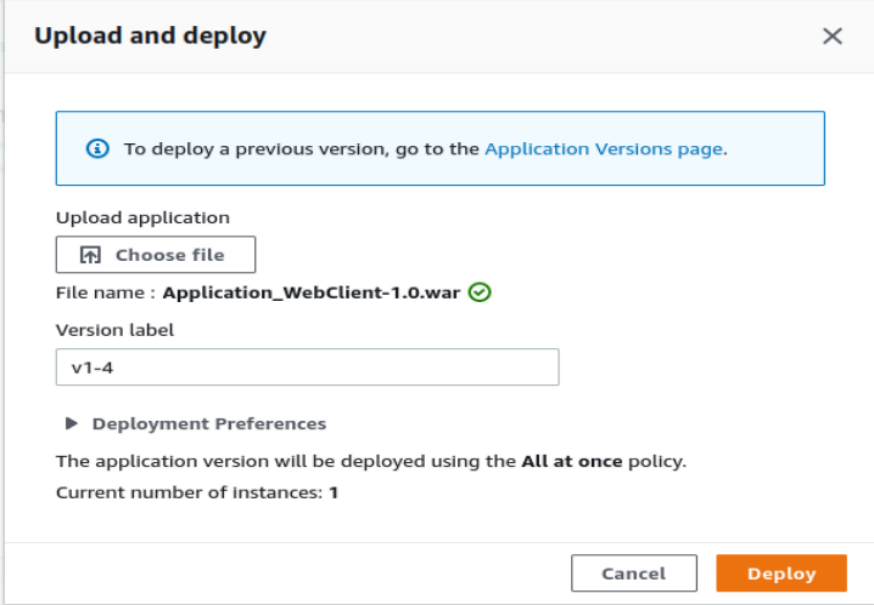
Imágen 4. Dashboard de nuestra aplicación en Elastic Beanstalk

AWS Elastic Beanstalk nos ofrece un nombre DNS para poder acceder a nuestra aplicación web `image-manager-pro.xxxxx.eu-west-1.elasticbeanstalk.com`. Para poder acceder de una forma más amigable hemos creado un registro DNS tipo CNAME en Amazon Route53 para apuntar a este nombre.

Subir nueva versión de la aplicación a AWS Elastic Beanstalk

Aplicaciones Distribuidas – GEI

Para subir una nueva versión clicamos en “Upload and deploy” y seleccionamos el war que nos genera Netbeans. En deployment preferences podemos especificar como queremos que se haga el despliegue en las instancias: todas de golpe, una a una, etc.. Podemos especificar una etiqueta para la versión subida.



The screenshot shows the 'Upload and deploy' dialog box. At the top, there is a close button (X). Below it, a blue box contains an information icon and the text: 'To deploy a previous version, go to the [Application Versions](#) page.' The 'Upload application' section has a 'Choose file' button. Below this, the 'File name' is 'Application_WebClient-1.0.war' with a green checkmark. The 'Version label' field contains 'v1-4'. The 'Deployment Preferences' section shows 'The application version will be deployed using the **All at once** policy.' and 'Current number of instances: 1'. At the bottom right, there are 'Cancel' and 'Deploy' buttons.

Imàgen 5. Formulario para subir una versión de código con el nuevo código adjuntado

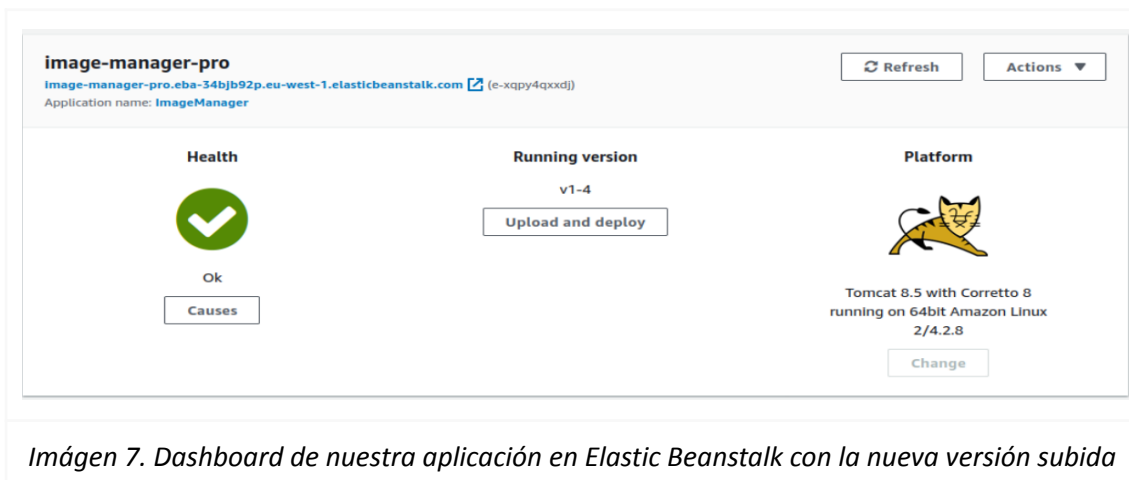
Clicamos en “Deploy” y empieza el despliegue.



The screenshot shows the AWS Elastic Beanstalk dashboard for the application 'image-manager-pro'. At the top, a blue banner states 'Elastic Beanstalk is updating your environment. To cancel this operation select **Abort Current Operation** from the **Actions** dropdown. [View Events](#)'. Below this, the application name 'image-manager-pro' is shown with its ARN and a link to the console. The dashboard is divided into three sections: 'Health' with a circular arrow icon and 'Ok' status, 'Running version' showing 'v1-3' and an 'Upload and deploy' button, and 'Platform' showing 'Tomcat 8.5 with Corretto 8 running on 64bit Amazon Linux 2/4.2.8' and a 'Change' button. There are 'Refresh' and 'Actions' buttons at the top right.

Imàgen 6. Dashboard de nuestra aplicación en Elastic Beanstalk durante el despliegue de una versión de la aplicación

Una vez terminado el despliegue, si todo ha ido bien, veremos como la aplicación está “OK” y la versión corriendo es la que hemos desplegado.



Imágen 7. Dashboard de nuestra aplicación en Elastic Beanstalk con la nueva versión subida

Existe un apartado dentro de AWS Elastic Beanstalk llamado “Application Versions” con todas las versiones que hemos desplegado correctamente. Desde este apartado podemos volver a desplegar una versión anterior y así volver a un punto anterior en nuestra aplicación.



Imágen 8. Dashboard de las distintas versiones de código subidas a AWS Elastic Beanstalk

Amazon Route53

Amazon Route 53 es un servicio de DNS (sistema de nombres de dominio) web escalable y de alta disponibilidad en la nube. Está diseñado para ofrecer a los desarrolladores y las empresas un método fiable y rentable para redirigir a los usuarios finales a las aplicaciones en Internet mediante la traducción de nombres legibles para las personas como `www.ejemplo.com` en direcciones IP numéricas como `192.0.2.1` que utilizan los equipos para conectarse entre ellos.

En la zona DNS pública `xpendack.net` hemos creado un registro tipo CNAME llamado `image-site.xpendack.net` que apunta al nombre DNS que nos da AW Elastic Beanstalk para el entorno desplegado.

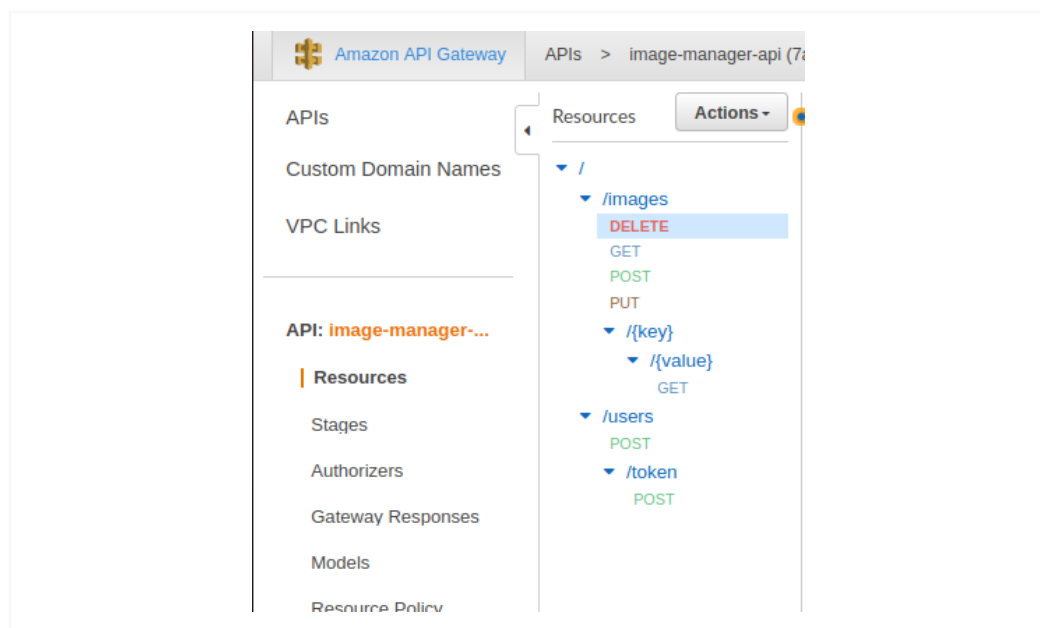
<input type="checkbox"/>	image-site.xpendack.net	CNAME	Simple	-	image-manager-pro.eba-dguu7ap9.eu-west-1.elasticbeanstalk.com
--------------------------	-------------------------	-------	--------	---	---

Imágen 9. Registro DNS creado en Route53

Backend

Amazon API Gateway

Amazon API Gateway es un servicio completamente administrado que facilita a los desarrolladores la creación, la publicación, el mantenimiento, el monitoreo y la protección de API a cualquier escala. Las API actúan como la "puerta de entrada" para que las aplicaciones accedan a los datos, la lógica empresarial o la funcionalidad de sus servicios de backend. Con API Gateway, podemos crear API RESTful y API WebSocket que permiten aplicaciones de comunicación bidireccional en tiempo real. API Gateway admite cargas de trabajo en contenedores y sin servidor, así como aplicaciones web.



Imágen 10. Detalle de los métodos creados en AWS API Gateway

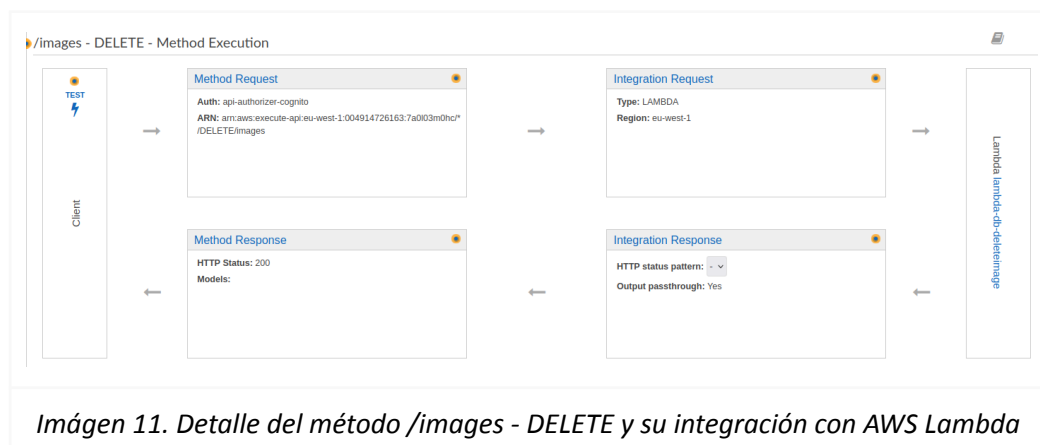
Hemos creado las siguientes operaciones en la API REST:

- /images
 - GET
 - Devuelve una lista con todas las imágenes que hay en el sistema.
 - Secured
 - Response
 - JSONArray: {list of images }
 - PUT
 - Modifica una imagen con la información facilitada.
 - Secured
 - Petición
 - JSON: {id: "", title: "title", description: "", keywords: "", author: "", creator: "", capture_date:"", filename:"",[optional] image_content: ""}
 - Response
 - JSON: { status: OK | ERROR, message: ""}
 - POST
 - Registra una imagen con la información facilitada.
 - Secured
 - Petición
 - JSON: {title: "title", description: "", keywords: "", author: "", creator: "", capture_date:"", filename:"", image_content: ""}
 - Response
 - JSON: { status: "success" | "fail" , msg: ""}
 - DELETE
 - Elimina la imagen del sistema dado su identificador.
 - Secured
 - Petición
 - JSON: {id: "id"}
 - Response
 - JSON: { status: "success" | "fail" , msg: ""}
- /images/{key}/{value}
 - GET
 - Busca las imágenes según key,value. Las keys pueden ser id, title,creator,author,keywords,creationdate.
 - Secured
 - Response
 - JSONArray: {list of images }
 -
- /users
 - POST
 - Crea un usuario en Amazon Cognito.
 - Not Secured
 - Petición
 - JSON: { username: "username", password: "password", email: "email" }
 - Response

- JSON: { status: "success" | "fail" , msg: "" }
- /users/token
 - POST
 - Comprueba que la contraseña especificada coincide con la del usuario especificado. Si coincide, genera un token y lo devuelve en la respuesta.
 - Not Secured
 - Petición
 - JSON: { username: "username", password: "password" }
 - Response
 - JSON: { status: "success" | "fail", id_token: "token" }

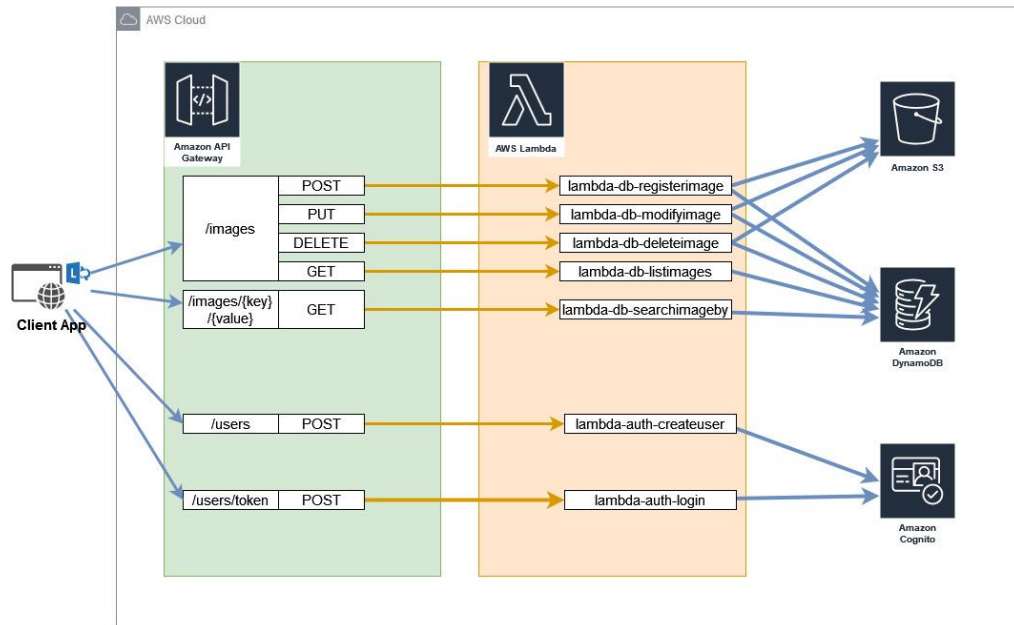
En la siguiente imagen se puede observar la integración del método /images - DELETE con la lambda **lambda-db-deleteimage**.

También se puede observar como en el atributo Auth tiene puesto como authorizer AWS Cognito para proteger las llamadas a la API.



Imágen 11. Detalle del método /images - DELETE y su integración con AWS Lambda

Cada una de las operaciones de la API REST está integrada con una Lambda. De esta forma tenemos modularizado el código por cada integración. La gestión de fallo y cambio es mucho más sencillo.



Imágen 12. Arquitectura Amazon API Gateway y la integración con AWS Lambda y demás servicios.

Para la seguridad de la API hemos configurado como Authorizer la user pool de Cognito. Cada vez que se efectúe una llamada en la API (y esta tenga activada la seguridad) se comprobará el header HTTP "Authorization" y se testeará el token para comprobar que sea válido.



Imagen 13. User Pool de AWS Cognito como Authorizer en la API Gateway

AWS Lambda

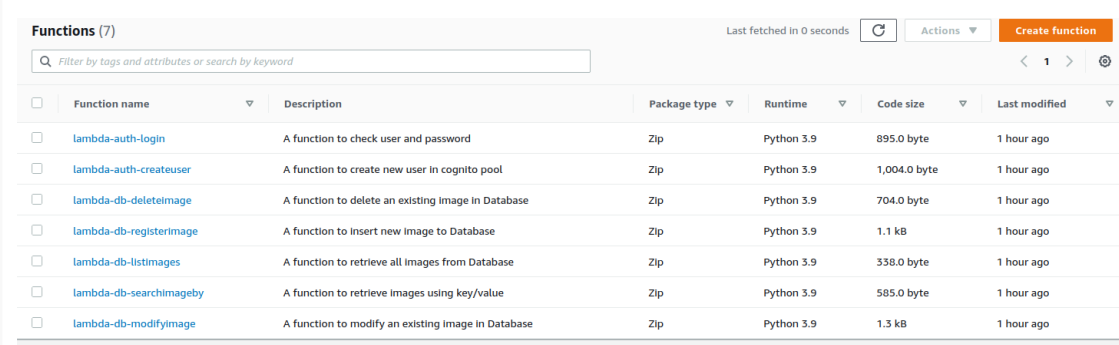
AWS Lambda es un servicio informático sin servidor y basado en eventos que nos permite ejecutar código para prácticamente cualquier tipo de aplicación o servicio backend sin

Aplicaciones Distribuidas – GEI

necesidad de aprovisionar o administrar servidores. Podemos activar Lambda desde más de 200 servicios de AWS y aplicaciones de software como servicio (SaaS).

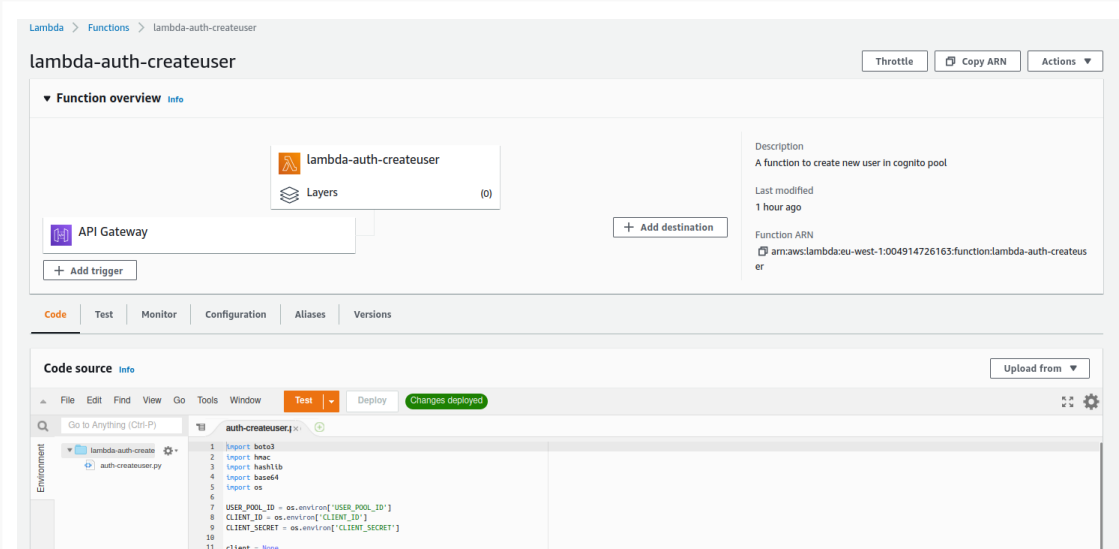
Hemos desarrollado una lambda para cada operación de la API Gateway. Están escritas en Python 3.9 ya que esta es la última versión estable.

Los códigos de las lambdas se encuentran en el directorio **Infra_AWS/source** del entregable.



<input type="checkbox"/>	Function name	Description	Package type	Runtime	Code size	Last modified
<input type="checkbox"/>	lambda-auth-login	A function to check user and password	Zip	Python 3.9	895.0 byte	1 hour ago
<input type="checkbox"/>	lambda-auth-createuser	A function to create new user in cognito pool	Zip	Python 3.9	1,004.0 byte	1 hour ago
<input type="checkbox"/>	lambda-db-deleteimage	A function to delete an existing image in Database	Zip	Python 3.9	704.0 byte	1 hour ago
<input type="checkbox"/>	lambda-db-registerimage	A function to Insert new Image to Database	Zip	Python 3.9	1.1 kB	1 hour ago
<input type="checkbox"/>	lambda-db-listimages	A function to retrieve all images from Database	Zip	Python 3.9	338.0 byte	1 hour ago
<input type="checkbox"/>	lambda-db-searchimageby	A function to retrieve images using key/value	Zip	Python 3.9	585.0 byte	1 hour ago
<input type="checkbox"/>	lambda-db-modifyimage	A function to modify an existing image in Database	Zip	Python 3.9	1.3 kB	1 hour ago

Imágen 14. Las lambdas creadas



The screenshot shows the 'lambda-auth-createuser' function details. It includes a 'Function overview' section with a diagram showing the function connected to an API Gateway. The 'Code source' section shows the Python code for the function, which imports boto3, hmac, hashlib, base64, and os, and uses environment variables for USER_POOL_ID, CLIENT_ID, and CLIENT_SECRET.

Imágen 15. Detalle de una de las Lambdas creadas (observar la integración con API Gateway)

El siguiente código muestra cómo insertamos una nueva imagen en el sistema.

Tenemos definidas varias variables de entorno como “table_name”, “bucketS3”, y “region” que están definidas en la lambda.

La lambda primero comprueba que estén todos los campos necesarios para dar de alta una imagen, sino devuelve error.

Después genera un identificador único para la imagen dado un timestamp. Este identificador nos servirá para identificar la imagen en la base de datos y en el fichero en S3. En dynamoDB no existen keys incrementales como en un base de datos SQL, por lo tanto, nosotros generamos un id único.

También generamos el campo storage_date en este momento.

Después guardamos la imagen en S3, pasándola del string en Base64 a fichero y le damos permisos de “public-read” para que sea consultable desde Internet.

Si, ha ido correctamente la subida a S3, la guardamos también en la base de datos.

```
import json
import boto3
import os
import base64

from boto3.dynamodb.conditions import Attr
from datetime import datetime, date

dynamodb = boto3.resource('dynamodb')
s3 = boto3.client('s3')
table_name = os.environ['TABLE_NAME']
bucketS3 = os.environ['BUCKET_S3']
region = os.environ['AWS_REGION']

def generate_unique_id():
    now = datetime.now()
    timestamp = str(int(round(datetime.timestamp(now))))
    return timestamp

def store_image_s3(image, filename, identifier):
    extension = filename.split('.')[1]
    new_filename = identifier + '.' + extension
    file_content = base64.b64decode(image)
    s3.put_object(Bucket=bucketS3, Key=new_filename,
    Body=file_content, ContentType='image/'+extension, ACL='public-read')
    return new_filename, None

def
store_image_dynamodb(id, title, description, keywords, author, creator, ca
pture_date, storage_date, filename):
    table = dynamodb.Table(table_name)
    table.put_item(
    Item={
        'id': id,
        'title': title,
```



```
        'description': description,
        'keywords': keywords,
        'author': author,
        'creator': creator,
        'capture_date': capture_date,
        'storage_date': storage_date,
        'filename': filename,
        'object_url':
f'https://{bucketS3}.s3.{region}.amazonaws.com/{filename}'
    }
    )
    return "OK", None

def lambda_handler(event, context):

    for field in ["title", "description",
"keywords", "author", "creator", "capture_date", "filename", "image_content"]]:
        if not event.get(field):
            return {
                'status': 'fail',
                'msg': f"{field} is not present"
            }

    title = event['title']
    description= event['description']
    keywords= event['keywords']
    author= event['author']
    creator = event['creator']
    capture_date = event['capture_date']
    filename = event['filename']
    image = event['image_content']

    identifier = generate_unique_id()
    storage_date = str(date.today())

    new_filename, msg = store_image_s3(image, filename, identifier)

    if msg != None:
        return {
            'status': 'fail',
            'msg': msg
        }
    }
```

```
    resp, msg =
store_image_dynamodb(identifier,title,description,keywords,author,cr
eator,capture_date,storage_date,new_filename)

    if msg != None:
    return {
        'status': "fail",
        'msg': "Image not registered!",
        'headers': {
            'Content-Type': 'application/json',
            'Access-Control-Allow-Origin': '*'
        }
    }
    response = {
        'status': "success",
        'msg': "Image registered!",
        'headers': {
            'Content-Type': 'application/json',
            'Access-Control-Allow-Origin': '*'
        }
    }

    return response
```

Código 1. Código de la Lambda para el registro de una Imagen en el sistema

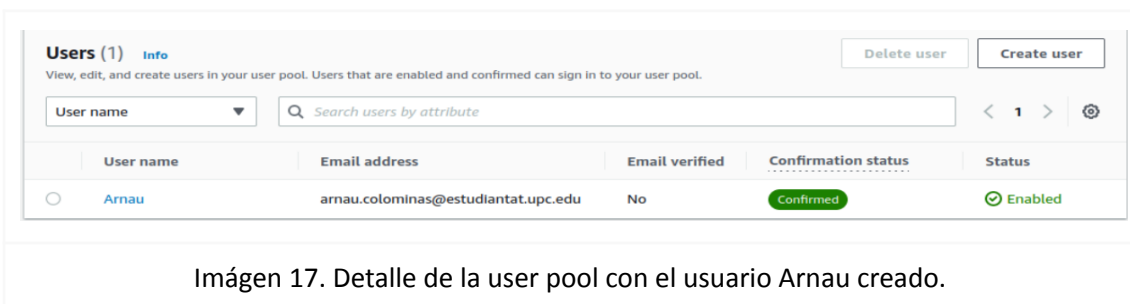
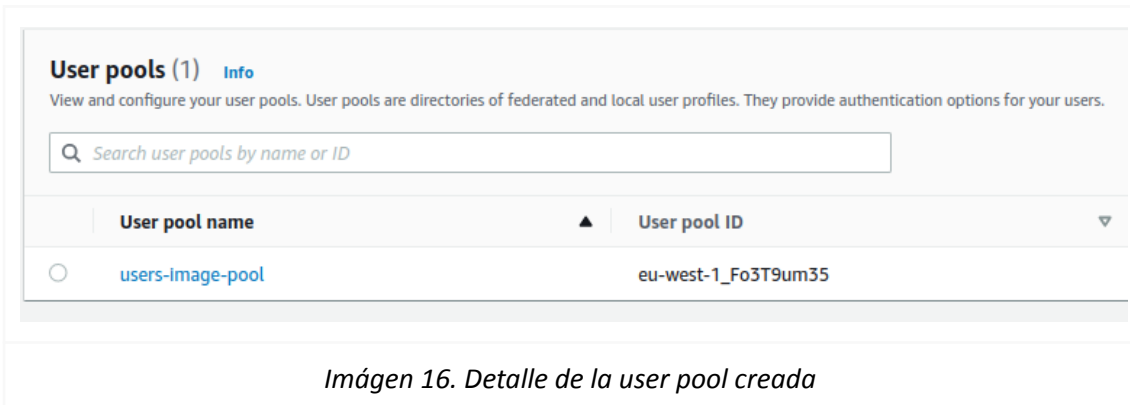
Amazon Cognito

Amazon Cognito nos permite incorporar de manera rápida y sencilla el registro, inicio de sesión y control de acceso de usuarios a aplicaciones web y móviles. Amazon Cognito cuenta con escalado para millones de usuarios y admite el inicio de sesión mediante proveedores de identidad social, como Apple, Facebook, Google y Amazon, así como con proveedores de identidad empresarial a través de SAML 2.0 y OpenID Connect.

Hemos creado un pool de usuarios llamado user-image-pool. En este user pool es donde las lambda crearán los nuevos usuarios y hará login para obtener el token de sesión.

En los usuarios es obligatorio que dispongan de dirección de correo electrónico.

También es donde las llamadas de la API comprobaran que el token de sesión pasado como header HTTP en Authorization sea correcto.



En el siguiente código mostramos como la lambda lambda-auth-login realiza en login contra el pool de usuarios creado anteriormente.

Dado un usuario y password realizamos una autenticación contra el user-pool creado anteriormente. El token obtenido lo devolvemos como respuesta, en caso de error devolvemos el error.

```
import boto3
import hmac
import hashlib
import base64
import os

USER_POOL_ID = os.environ['USER_POOL_ID']
CLIENT_ID = os.environ['CLIENT_ID']
CLIENT_SECRET = os.environ['CLIENT_SECRET']

client = None

def get_secret_hash(username):
    msg = username + CLIENT_ID
    digest = hmac.new(str(CLIENT_SECRET).encode('utf-8'),
msg=str(msg).encode('utf-8'), digestmod=hashlib.sha256).digest())
```

```
    dec = base64.b64encode(digest).decode()
    return dec

def initiate_auth(username, password):
    try:
        resp = client.admin_initiate_auth(
            UserPoolId=USER_POOL_ID,
            ClientId=CLIENT_ID,
            AuthFlow='ADMIN_NO_SRP_AUTH',
            AuthParameters={
                'USERNAME': username,
                'SECRET_HASH': get_secret_hash(username),
                'PASSWORD': password
            },
            ClientMetadata={
                'username': username,
                'password': password
            })
    except client.exceptions.NotAuthorizedException as e:
        return None, "The username or password is incorrect"
    except client.exceptions.UserNotFoundException as e:
        return None, "The username or password is incorrect"
    except Exception as e:
        print(e)
        return None, "Unknown error"
    return resp, None

def lambda_handler(event, context):
    global client
    if client == None:
        client = boto3.client('cognito-idp')

    for field in ["username", "password"]:
        if not event.get(field):
            return {
                'status': 'fail',
                'msg': f"{field} is not present"
            }

    username = event['username']
    password = event['password']

    resp, msg = initiate_auth(username, password)
```

```
if msg != None:
    return {
        'status': 'fail',
        'msg': msg
    }

response = {
    'status': 'success',
    'id_token': resp['AuthenticationResult']['IdToken']
}

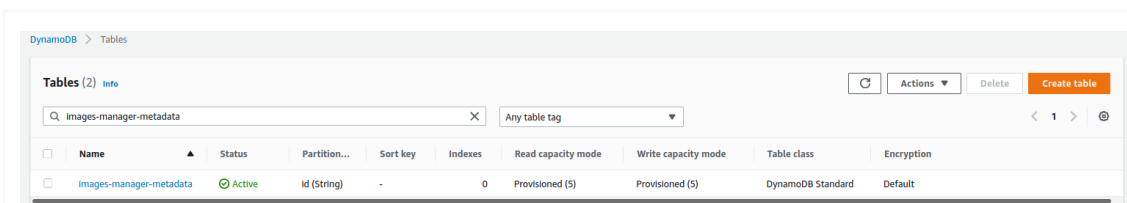
return response
```

Código 2. Código de la Lambda para el login de un usuario y el retorno del token

Amazon DynamoDB

Amazon DynamoDB es una base de datos NoSQL de clave de valor sin servidor completamente administrada que está diseñada para ejecutar aplicaciones de alto rendimiento a cualquier escala. DynamoDB ofrece seguridad integrada, copias de seguridad continuas, replicación automatizada en varias regiones, almacenamiento de caché en memoria y herramientas de exportación de datos

Hemos creado la tabla DynamoDB image-manager-table. En esta tabla se almacenan los metadatos de las imágenes.



Name	Status	Partition...	Sort key	Indexes	Read capacity mode	Write capacity mode	Table class	Encryption
images-manager-metadata	Active	Id (String)	-	0	Provisioned (5)	Provisioned (5)	DynamoDB Standard	Default

Imágen 18. Detalle de la tabla image-manager-table creada

La API Gateway mediante las lambdas crea, modifica y elimina las entradas en la tabla image-manager-metada



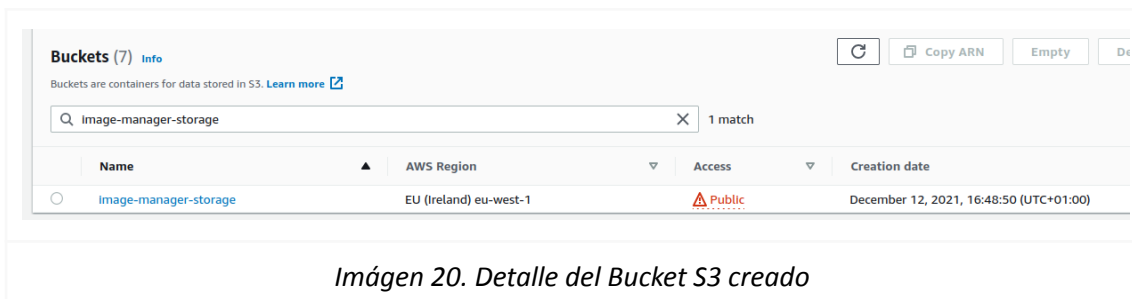
id	author	capture_...	creator	description	filename	keywords
1639560767	Marc	2021-12-14	Arnau	desc1	163956076...	key1,key2

Imagen 19. Detalle de la entrada en la tabla de una imagen (metadatos).

Amazon S3

Amazon Simple Storage Service (Amazon S3) es un servicio de almacenamiento de objetos que ofrece escalabilidad, disponibilidad de datos, seguridad y rendimiento líderes en el sector. Clientes de todos los tamaños y sectores pueden almacenar y proteger cualquier cantidad de datos para prácticamente cualquier caso de uso: las aplicaciones nativas en la nube y las aplicaciones móviles. Con clases de almacenamiento rentables y características de administración fáciles de utilizar, puede optimizar los costos, organizar los datos y configurar los controles de acceso precisos con objeto de satisfacer requisitos empresariales, organizativos y de conformidad específicos.

Hemos creado un Bucket S3 llamado image-manager-storage. Este bucket tiene permisos tipo Public, es decir es accesible desde Internet. Este bucket nos sirve como storage para los ficheros de las imágenes, es nuestro “disco”.



Imágen 20. Detalle del Bucket S3 creado

La API Gateway mediante las lambdas crea y elimina las imágenes en este Bucket. La API recibe la imagen en formato string Base64 y lo convierte en fichero.

Cuando el fichero se guarda en el bucket, se le dan permisos de lectura desde Internet para que sea consultable desde la aplicación web.

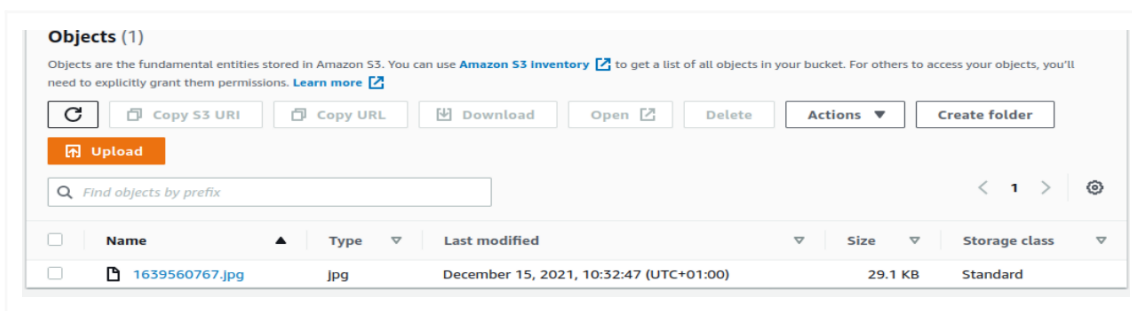


Imagen 21. Detalle de una imagen almacenada en el Bucket S3 creado.

Escalabilidad del diseño

La aplicación es totalmente escalable. Los servicios Serverless lo son por definición.

Además hemos añadido los thresholds en AWS Elastic Beanstalk para ir añadiendo o quitando instancias según el consumo de CPU.

Bibliografía consultada

<https://www.genbeta.com/desarrollo/que-serverless-que-adaptarlo-desarrollo-tu-proxima-aplicacion>

<https://registry.terraform.io/providers/hashicorp/aws/latest/docs>

<https://aws.amazon.com/es/what-is-aws/>

https://es.wikipedia.org/wiki/Amazon_Web_Services

<https://aws.amazon.com/es/elasticbeanstalk/>

<https://aws.amazon.com/es/cognito/>

<https://aws.amazon.com/es/api-gateway/>

<https://betterprogramming.pub/secure-aws-api-gateway-with-amazon-cognito-and-aws-lambda-535e7c9ffea1>

<https://docs.aws.amazon.com/elasticloadbalancing/latest/application/sticky-sessions.html>

<https://aws.amazon.com/es/route53/>

<https://aws.amazon.com/es/dynamodb/>

<https://aws.amazon.com/es/lambda/>

<https://aws.amazon.com/es/s3/>

<https://stackoverflow.com/questions/51962633/return-to-servlet-in-clustered-environment>

<https://stackoverflow.com/questions/1303533/access-environment-variable-from-java-servlet>

<https://docs.oracle.com/javase/8/docs/technotes/guides/security/jsse/JSSERefGuide.html#StandardAPI>