UPLOAD SERVICE

NFTMARKET

Fundamentos de Ingeniería del Software para Sistemas Cloud

Miembros:

Reyes Blasco Cuadrado

Juan Fernández Fernández

Índice

Índice de ilustraciones

Acabado del proyecto

Nivel de acabado

El microservicio de Upload aspira al nivel de acabado del nivel de hasta **9 puntos**. Para ello se han implementado todos los requisitos del nivel hasta 5 puntos, además de 5 características del microservicio avanzado y 3 de la aplicación basada en microservicios avanzada, además de la básica.

El objetivo de este microservicio es gestionar las imágenes de los productos, encargándose de realizar las peticiones correspondientes al almacenamiento de datos en Cloud, en nuestro caso, Google Photos. De igual forma, debe ofrecer al resto de servicios, las imágenes y URLs correspondientes.

A continuación, se listan las características implementadas:

- **Nivel hasta 5 puntos**:
  - El backend debe ser un API REST tal como se ha visto en clase implementando al menos los métodos GET, POST, PUT y DELETE y devolviendo un conjunto de códigos de estado adecuado.
  - La API debe tener un mecanismo de autenticación.
  - Debe tener un frontend que permita hacer todas las operaciones de la API.
  - Debe estar desplegado en la nube y ser accesible en una URL.
  - La API que gestione el recurso también debe ser accesible en una dirección bien versionada.
  - Se debe tener una documentación de todas las operaciones de la API incluyendo las posibles peticiones y las respuestas recibidas.
  - Debe tener persistencia utilizando MongoDB u otra base de datos no SQL.
  - Validación de los datos antes de almacenarlos en la base de datos (por ejemplo, haciendo uso de mongoose).
  - Se debe utilizar gestión del código fuente y mecanismos de integración continua.
  - Debe haber definida una imagen Docker del proyecto.
  - Debe haber pruebas unitarias implementadas en Javascript para el código del backend utilizando Jest(el usado en los ejercicios)o Mocha y Chai o similar.
  - Debe haber pruebas de integración con la base de datos.
- **Nivel hasta 9 puntos**:
  - Microservicio avanzado:
    - Implementar un frontend con rutas y navegación.
    - Uso del patrón materialized view para mantener internamente el estado de otros microservicios.
    - Tener el APIREST documentado con swagger.
    - Implementación de un mecanismo de autenticación basado en JWT o equivalente.
    - Utilizar algún otro tipo de almacenamiento de datos en cloud como Amazon S3.
  - Aplicación basada en microservicios:
    - Interacción completa entre todos los microservicios de la aplicación integrando información.
    - Tener un front end común que integre los front ends de cada uno de los microservicios.
    - Hacer uso de un sistema de comunicación asíncrono mediante un sistema de cola de mensajes para todos los microservicios.
    - Implementación de un mecanismo de autenticación homogéneo para todos los microservicios.

Implementación del código

En este apartado, se especifica y justifica la implementación de cada uno de los requisitos especificados en el apartado anterior. Para ello, se divide el mismo en diferentes subapartados dependiendo del nivel de acabado.

Nivel hasta 5 puntos

— **El backend debe ser un API REST tal como se ha visto en clase implementando al menos los métodos GET, POST, PUT y DELETE y devolviendo un conjunto de códigos de estado adecuado.**

El backend está formado por 5 llamadas destinadas a mostrar, listar (GET), crear (POST), modificar (PUT) y borrar (DELETE) una imagen o asset. Esto se puede visualizar en el archivo server.js en la raíz del proyecto.

```
// CREAR ASSET
app.post(BASE_API_PATH + "/asset", [authorizedClient], async (req, res) => {
    console.log(Date() + " - POST /asset");

// LISTAR ASSETS
app.get(BASE_API_PATH + "/asset", authorizedClient, (req, res) => {
    console.log(Date() + " - GET /asset");
// MODIFICAR ASSET
app.put(BASE_API_PATH + "/asset/:id", authorizedClient, async (req, res) => {
    console.log(Date() + " - UPDATE /asset");
// OBTENER UN ASSET
app.get(BASE_API_PATH + "/asset/:id", authorizedClient, (req, res) => {
    console.log(Date() + " - GET /assets/:id");
// BORRAR ASSET
app.delete(BASE_API_PATH + "/asset/:id", authorizedClient, async (req, res) => {
    console.log(Date() + " - DELETE /assets/:id");
```

Ilustración 1. Definición de EndPoints de Backend

— **La API debe tener un mecanismo de autenticación.**

Se ha hecho uso de JWT para la autenticación del usuario. Para realizar la validación de este, se ha usado la función *authorizedClient*, que se encuentra definida en el archivo *middlewares/auhorized-roles.js*.

En la siguiente imagen se muestra la implementación de uno de ellos, en la que se obtiene la información de la cabecera header y se verifica que el jwt y el rol sean válidos para la petición realizada.

```
const authorizedClient = (req, res, next) => {
  try {
    const token = req.header("Authorization");

    if (!token) {
      return res.status(401).json({
        msg: "Token is not provided",
      });
    }
    const payload = jwt.verify(
      token.replace("Bearer ", ""),
      process.env.SECRET_KEY
    );

    if (!verifyRole(payload.role, CLIENT)) {
      return res.status(403).json({
        msg: "Unauthorized role",
      });
    }

    req.id = payload.id;

    next();
  } catch (e) {
      console.log(e);
    return res.status(401).json({
      msg: "The provided JWT is malformed",
    });
  }
};
```

Ilustración 2. Definición de authorizedClient del JWT

— **Debe tener un frontend que permita hacer todas las operaciones de la API.**

El front end está compuesto por dos vistas principales, el listado de assets y la vista individual. En el listado se hacen uso de las funciones list y create, mientras que en la vista individual se hace uso de las funciones show, update y delete.

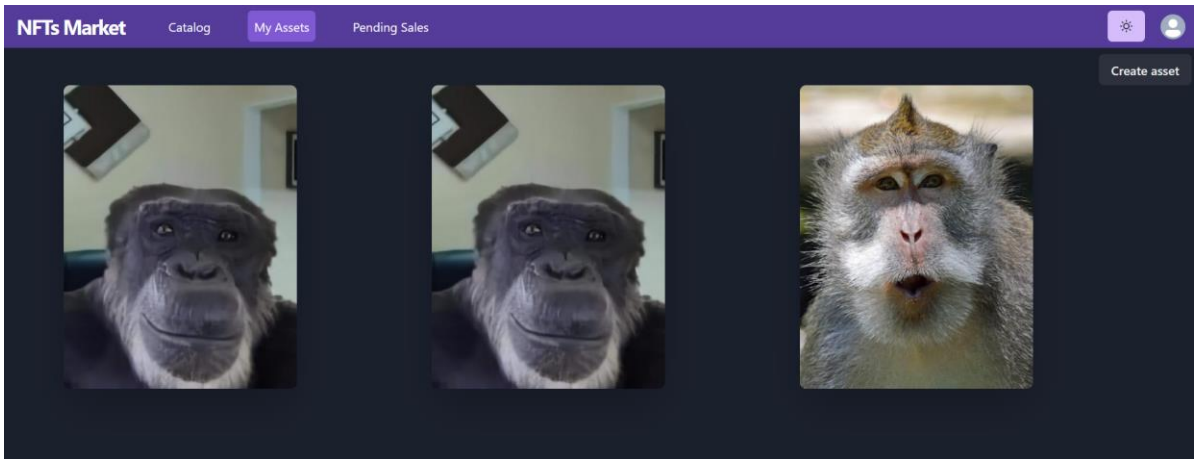


Ilustración 3. Listado de assets en FrontEnd

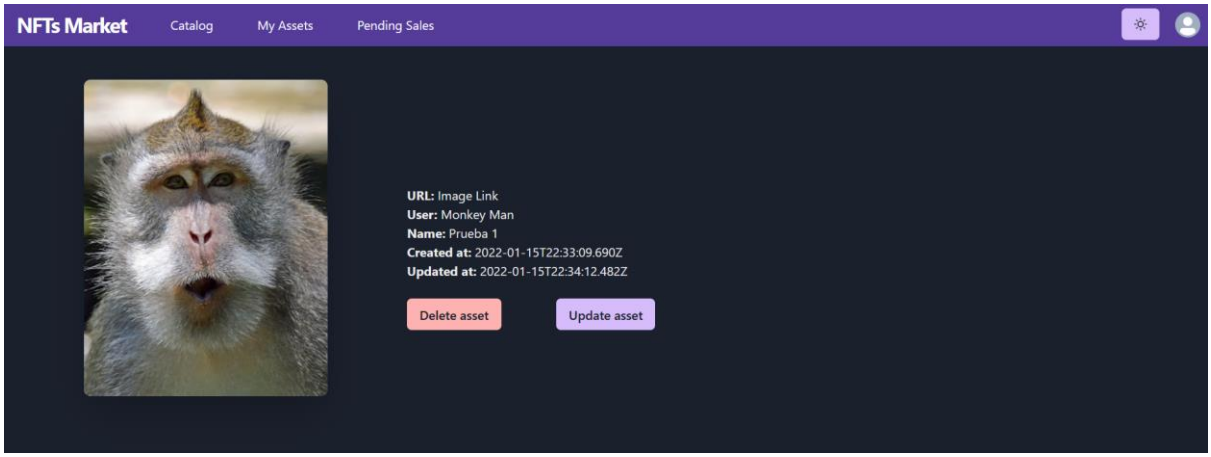


Ilustración 4. Vista individual de asset en FrontEnd

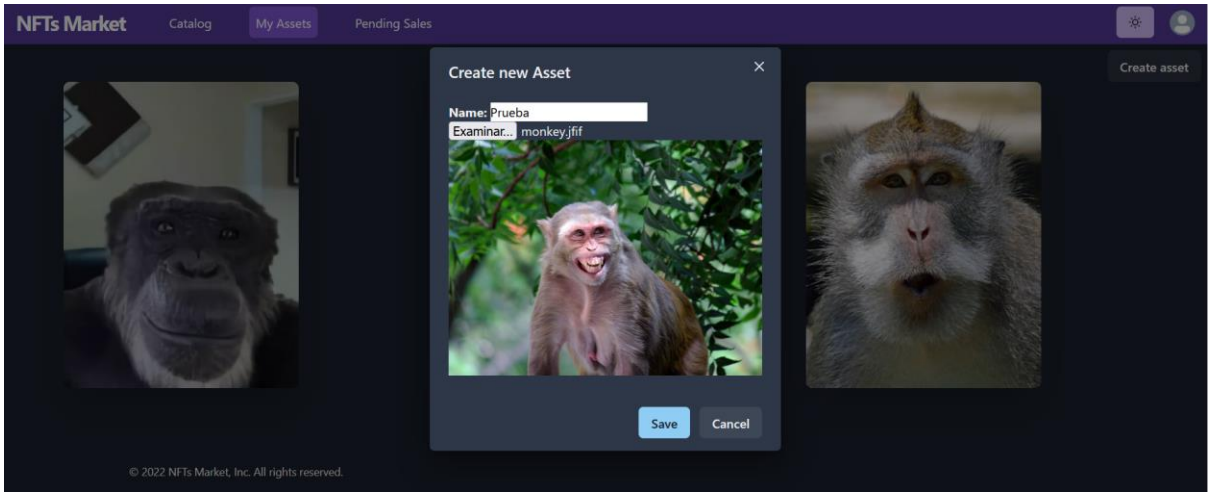


Ilustración 5. Creación de un asset en FrontEnd

— **Debe estar desplegado en la nube y ser accesible en una URL.**

Los sistemas se encuentran desplegados en los siguientes enlaces:

BackEnd: https://api-reyblacua.cloud.okteto.net

FrontEnd: https://nftsmarket.netlify.app/

— **La API que gestione el recurso también debe ser accesible en una dirección bien versionada.**

La API se encuentra versionada mediante la cadena "/api/v1" en el fichero server de la raíz del proyecto.

```
var BASE_API_PATH = "/api/v1";
```

— **Se debe tener una documentación de todas las operaciones de la API incluyendo las posibles peticiones y las respuestas recibidas.**

Se ha realizado la definición de API en Swagger. Se puede acceder a ella a través del enlace de swagger o a través del backend desplegado.

Swagger: https://app.swaggerhub.com/apis-docs/reyblacua/UploadService/1.2

Backend: https://api-reyblacua.cloud.okteto.net

— **Debe tener persistencia utilizando MongoDB u otra base de datos no SQL.**

Los datos se almacenan en MongoDB y se hace uso de Mongoose para su conexión y gestión de clases. Esto se puede ver en la definición de los modelos en la carpeta *models*, y en la conexión con la base de datos en el archivo db.js.

```
const mongoose = require('mongoose');
const googlePhotos= require('./googlePhotos/googlePhotosService')

var DB_URL = ('mongodb://localhost:27017/test');



if(process.env.MONGO_HOSTNAME!=undefined){
    DB_URL = ('mongodb://'+process.env.MONGO_HOSTNAME+':27017/test');
}else{
    DB_URL = (process.env.MONGO_URL|| 'mongodb://localhost:27017/test');
}

const dbConnect = function() {
    const db = mongoose.connection;
    googlePhotos.initializeGoogleCloud();
    db.on('error', console.error.bind(console, 'connection error: '));
    return mongoose.connect(DB_URL, { useNewUrlParser: true });
}

module.exports = dbConnect;
```

Ilustración 6. Conexión del servidor con MongoDB

— **Validación de los datos antes de almacenarlos en la base de datos.**

La validación de los datos se ha realizado mediante los schemas de Mongoose y, a su vez, se han implementado de forma manual, para ofrecer mayor seguridad en la subida y modificación de los datos. Esto se puede ver en la carpeta *models* y en el archivo *server*.

```
var mongoose = require('mongoose'),
  Schema = mongoose.Schema;

const schema = new Schema({
  file: {
    type:String,
    required:[true, "The file can not be empty"],
  },
  name: {
    type:String,
    required:[true, "The name can not be empty"]
  },
  user: {
    type: String,
    required:[true, "The user can not be empty"]
    }
},
  {
    timestamps: true
  });

module.exports = mongoose.model('Asset', schema);
```

Ilustración 7. Validación mediante Mongoose

```
if (typeof req.body.file !== 'string' || !req.body.file instanceof String) {
    return res.status(StatusCodes.BAD_REQUEST).json("File must be a string.");
} else if (req.body.file.match(/^ *$/) !== null) {
    return res.status(StatusCodes.BAD_REQUEST).json("File can't be whitespace or empty.");
} else if (typeof req.body.name !== 'string' || !req.body.name instanceof String) {
    return res.status(StatusCodes.BAD_REQUEST).json("Name must be a string.");
} else if (req.body.name.match(/^ *$/) !== null) {
    return res.status(StatusCodes.BAD_REQUEST).json("Name can't be whitespace or empty.");
} else if (typeof req.body.user !== 'string' || !req.body.user instanceof String) {
    return res.status(StatusCodes.BAD_REQUEST).json("User must be a string.");
} else if (req.body.user.match(/^ *$/) !== null) {
    return res.status(StatusCodes.BAD_REQUEST).json("User can't be whitespace or empty.");
}
```

Ilustración 8. Validación manual de los datos de entrada

— **Se debe utilizar gestión del código fuente y mecanismos de integración continua.**

Se ha hecho uso de GitHub, y se han definido los pipelines necesarios para github actions en el que se ejecutan las pruebas y se despliega el sistema una vez han pasado estas. Estos pipelines se encuentran en la carpeta *.github/workflows* del repositorio:

Enlace al repositorio: https://github.com/NFTsMarket/Upload-Service

Enlace al pipeline: https://github.com/NFTsMarket/Upload-Service/tree/develop/.github/workflows

— **Debe haber definida una imagen Docker del proyecto.**

En el proyecto, se puede encontrar la imagen de Docker definida en el archivo *Dockerfile*, y el fichero *docker-compose* asociado al mismo. Se adjuntan los enlaces a ambos archivos a continuación:

Dockerfile: https://github.com/NFTsMarket/Upload-Service/blob/develop/Dockerfile

Docker-compose: https://github.com/NFTsMarket/Upload-Service/blob/develop/docker-compose.yml

— **Debe haber pruebas unitarias implementadas en Javascript para el código del backend utilizando Jest(el usado en los ejercicios)o Mocha y Chai o similar.**

Se han implementado un total de 33 pruebas unitarias sobre los endpoint comentados anteriormente, utilizando casos de éxito y error, concretamente la ejecución correcta de las llamadas, fallo por validación y errores producidos en la base datos. Estas pruebas se pueden encontrar en el siguiente enlace: https://github.com/NFTsMarket/Upload-Service/blob/develop/tests/server.test.js

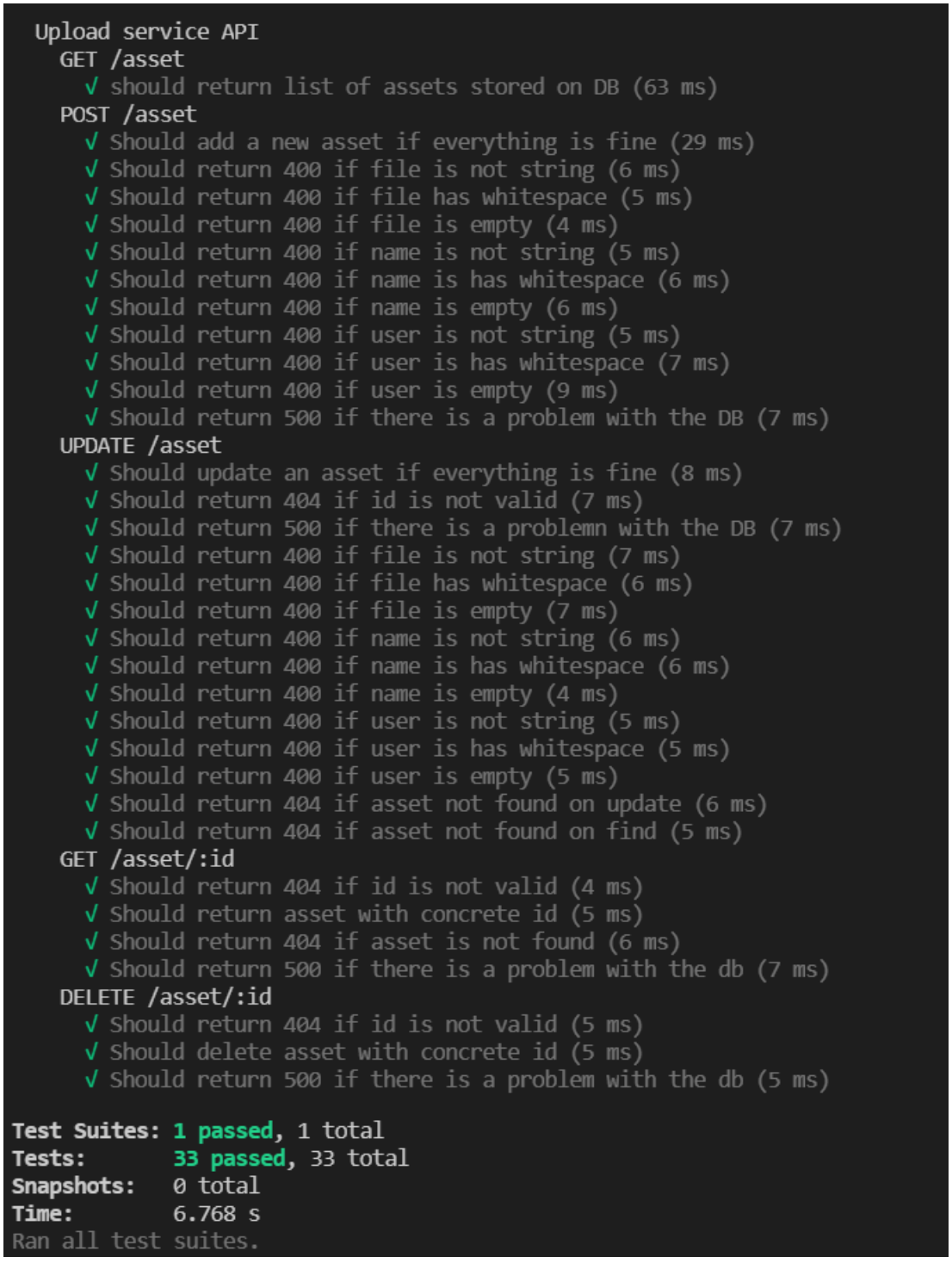


Ilustración 9. Ejecución de las pruebas unitarias

— **Debe haber pruebas de integración con la base de datos.**

Se han realizado las pruebas de integración correspondientes a la incorporación y consulta de datos en la base de datos. Estas pruebas se pueden encontrar en el fichero: https://github.com/NFTsMarket/Upload-Service/blob/develop/tests/integration/integration-db.test.js

Nivel hasta 9 puntos

Microservicio avanzado

— **Implementar un front end con rutas y navegación.**

El front end se encuentra implementado mediante la librería next, que permite el enrutado de la aplicación de forma sencilla. Las páginas implementadas por el servicio de upload se corresponden con los siguientes enlaces:

Listado de assets: https://nftsmarket.netlify.app/assets

Vista individual de assets: https://nftsmarket.netlify.app/assets/{id}

Desde el listado se puede acceder a la vista individual de un asset al pulsar encima de una de las imágenes. Esto hará que la aplicación cargue la vista individual con los datos asociados a la imagen con el id especificado en la url.

Si el usuario no se ha identificado previamente, el sistema redirigirá al usuario a la página de catálogo. Para poder acceder a cualquiera de las dos URLs definidas, el usuario debe estar autentificado para poder mostrar los assets que haya subido a la plataforma.

— **Uso del patrón materialized view para mantener internamente el estado de otros microservicios**

Se hace uso del patrón materialized view debido a la implementación de la integración mediante PubSub. Al tener las suscripciones por parte del microservicio *Autenticación*, recibimos los datos de los usuarios en las llamadas de creación, actualización y borrado. Estos datos son almacenados en una colección *User* en nuestra base de datos, la cual utilizamos para no tener que realizar consultas al microservicio de *Autenticación*, ya que los datos se actualizarán a partir de lo que se indique en la suscripción que corresponda.

La información de *User* se usa en la muestra de los datos de la imagen, en la que se proporciona la información del usuario que posee la imagen.

En la siguiente imagen, se puede ver la suscripción correspondiente a la creación de *User*. En la misma se puede ver como se ejecuta la creación del usuario en mongoose.

```
// On Created user
this.pubsub
.subscription("upload-created-user")
.on("message", async (message) => {
    console.log("Receiving...");
    console.log(JSON.parse(message.data.toString()));
    const user = JSON.parse(message.data.toString());
   if(user!=null){
     try{
       await User.create(user);
     }catch(e){
       console.log(e);
     }
   }
    message.ack();
});
```

Ilustración 10. Suscripción de la creación de usuarios

Se puede ver un ejemplo del uso de los datos de User en el método GET de *Asset*. En este, se realiza la consulta del *Asset* y, posteriormente, se obtiene el *User* con el id del usuario del *Asset*.

```
// OBTENER UN ASSET
app.get(BASE_API_PATH + "/asset/:id", authorizedClient, (req, res) => {
    console.log(Date() + " - GET /assets/:id");
    if (!ObjectId.isValid(req.params.id)) {
        return res.status(StatusCodes.NOT_FOUND).json("An asset with that id could not be found, since it's not a valid id.");
    }

    var filter = { _id: req.params.id };
    Asset.findOne(filter, async function (err, asset) {
        if (err) {
            return res.status(StatusCodes.INTERNAL_SERVER_ERROR).json(err);
        } else if (asset) {
            var filter2 = { id: asset.user };
            var user = await User.find(filter2);
            var token = await googlePhotos.get_access_token_using_saved_refresh_token();
            var googlePhotosResponse = await googlePhotos.getAsset(token, asset.file);
            asset._doc["image"] = googlePhotosResponse;
            asset._doc["user"] = user[0];
            return res.status(StatusCodes.OK).json(asset);
        } else {
            return res.status(StatusCodes.NOT_FOUND).json("An asset with that id could not be found.");
        }
    });
});
```

Ilustración 11. Consulta de un asset junto al usuario correspondiente

— **Tener el APIREST documentado con swagger.**

Como se ha comentado anteriormente, se ha definido la API REST en Swagger. Adjuntamos de nuevo los enlaces disponibles:

Swagger: https://app.swaggerhub.com/apis-docs/reyblacua/UploadService/1.2

Backend: https://api-reyblacua.cloud.okteto.net

— **Implementación de un mecanismo de autenticación basado en JWT o equivalente.**

Se ha hecho uso de JWT para la autenticación del usuario. Para realizar la validación de este, se ha usado la función *authorizedClient*, que se encuentra definida en el archivo *middlewares/auhorized-roles.js*, como se mencionó anteriormente.

— **Utilizar algún otro tipo de almacenamiento de datos en cloud como Amazon S3.**

Se ha hecho uso de Google Photos como almacenamiento de datos en cloud. En el directorio *googlePhotos* del proyecto se establece la conexión y definen las peticiones a la API de Google Photos. Este archivo se puede encontrar en la siguiente dirección: https://github.com/NFTsMarket/Upload-Service/blob/develop/googlePhotos/googlePhotosService.js

Aplicación basada en microservicios

— **Interacción completa entre todos los microservicios de la aplicación integrando información.**

Se ha realizado la integración entre los servicios mediante Pub-Sub, el cuál explicaremos en el apartado de sistema de comunicación asíncrono.

— **Tener un front end común que integre los front ends de cada uno de los microservicios. Cada pareja debe ocuparse, al menos, de la parte específica de su microservicio en el front end común.**

Se ha desarrollado un front end común. Este se encuentra desplegado en la dirección: https://nftsmarket.netlify.app/

La sección de “My Assets” es la perteneciente a la gestión de las imágenes del usuario.

— **Hacer uso de un sistema de comunicación asíncrono mediante un sistema de cola de mensajes para todos los microservicios. Si no es para todos, debe justificarse de forma razonada.**

Como se ha comentado anteriormente, se ha hecho uso de Pub Sub. La implementación del envío de datos se encuentra en el directorio *controllers/serverController*: https://github.com/NFTsMarket/Upload-Service/blob/develop/controllers/serverController.js

Las suscripciones a los servicios se encuentran en el directorio *models/subscriptions*: https://github.com/NFTsMarket/Upload-Service/blob/develop/models/subscriptions.js

Se han realizado suscripciones a los servicios *Catalogue* y *Authentication*. En cuanto a *Authentication*, debemos obtener los datos del usuario cada vez que se cree, modifique o se borre uno. En el caso de *Catalogue*, se utiliza para realizar el cambio de usuario cuando se produce una compra.

En este último caso, *Catalogue* consume la petición de *Buy* y lanza un mensaje con el usuario para que el servicio de *Upload* pueda consumir este y cambiar el propietario de la imagen.

```
async function sendMessageCreatedAsset (req,googlePhotosResponseId) {
  try {
    const data = req;

    var token = await googlePhotos.get_access_token_using_saved_refresh_token();
    var googlePhotosResponse = await googlePhotos.getAsset(token, googlePhotosResponseId);

    const asset={
      id:data._id,
      name:data.name,
      user:data.user,
      file:googlePhotosResponse.baseUrl
    }
    await publishPubSubMessage("created-asset", asset);

    console.log("Message sent to PubSub");
  } catch (e) {
    console.log(e);
    res.status(500).send(e);
  }
};
```

Ilustración 12. Definición de envío de datos mediante PubSub

```
// On changed user
this.pubsub
.subscription("upload-updated-product")
.on("message", async (message) => {
    console.log("Receiving...");
    console.log(JSON.parse(message.data.toString()));
    const product = JSON.parse(message.data.toString());

    if(product!=null){
     const body={
       user:product.owner
     }
     try{
       var filter = { _id: product.picture };
       Asset.findOneAndUpdate(filter, body, function (err, doc) {
         if (!doc) {
             console.log("An asset with that id could not be found.");
         }
       });

     }catch(e){
       console.log(e);
     }
   }

    message.ack();
});
```

Ilustración 13. Definición de la suscripción de actualización de producto

— **Implementación de un mecanismo de autenticación homogéneo para todos los microservicios.**

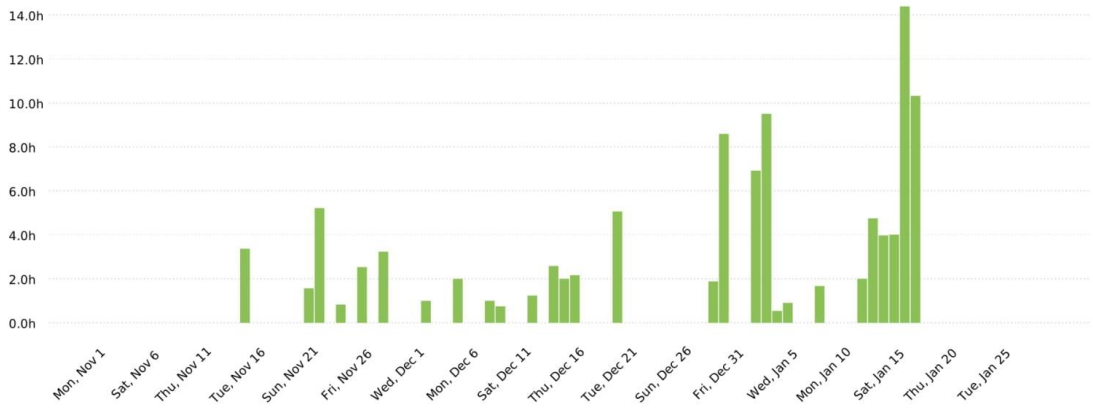
Se ha implementado la autenticación mediante JWT en todos los servicios del sistema. El ejemplo de implementación de este se puede observar en los diferentes apartados anteriores.

Análisis de esfuerzo

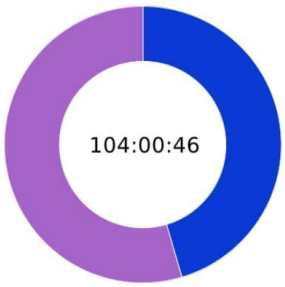
Summary report

11/01/2021 - 01/27/2022

Total: 104:00:46



User



juaferfer11	56:48:01	54.61%
Reyblacua	47:12:45	45.39%

Description

Configuración de pruebas automáticas de backend	00:24:00	0.38%
Integrar listado de backend #19	03:32:24	3.40%
Integrar backend con autenticación	03:44:00	3.59%
Configurar integración con google photos con variables de entorno	03:04:00	2.95%
Añadir variables de entorno en github actions y okteto pipeline	05:30:00	5.29%

FIS Created with Clockify 1

Ilustración 14. Resumen del esfuerzo dedicado