Universidad de Sevilla

Escuela Internacional de Posgrado

NFTsMarket – Buy Service

Máster en Ingeniería del Software: Cloud, Datos y Gestión TI

Fundamentos de Ingeniería del Software para Sistemas Cloud

Curso 2021 – 2022

Documentación del servicio en SWAGGER

Integración continua del microservicio

Repositorio del microservicio

Despliegue del microservicio en Okteto

Repositorio común del FrontEnd

Despliegue del sistema (FrontEnd)

Integración continua FrontEnd

Fecha	Versión
18/01/2022	1.0

Autores
Sánchez León, Sergio
Sola Espinosa, Fernando Luis

Índice de contenido

1. Nivel de acabado elegido

Optamos al nivel de acabado **Hasta 9 puntos**, habiendo seleccionado las 5 características de microservicio avanzado y las 5 (dos más de las que se requiere para el nivel 9 puntos) de aplicación basada en microservicios avanzada que se detallan a continuación.

1.1. Características de microservicio avanzado que gestione un recurso elegidas

- Implementar un frontend con rutas y navegación.
- Uso del patrón materialized view para mantener internamente el estado de otros microservicios.
- Consumo de algún API externa (distinta de las de los grupos de práctica).
- Tener el API REST documentado con swagger.
- Implementación de un mecanismo de autenticación basado en JWT o equivalente.

1.2. Características de aplicación basada en microservicios avanzada elegidas

- 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.
- Hacer uso de un API Gateway con funcionalidad avanzada como un mecanismo de throttling o de autenticación.
- Definición de un customer agreement para la aplicación en su conjunto.
- 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.
- Implementación de un mecanismo de autenticación homogéneo para todos los microservicios.

2. Justificación de la consecución de los requisitos

2.1. Microservicio básico que gestione un recurso

2.1.1. 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

Toda la funcionalidad de nuestra api se encuentra en el archivo *server.js* de nuestro repositorio donde se pueden encontrar todos los métodos básicos para la entidad *purchase*:

```
// GET purchases API method
app.get(BASE_API_PATH + "/purchase/", authorizedClient, (req, res) => {
    console.log(Date() + " - GET /purchase/");


// GET a specific purchase API method
app.get(BASE_API_PATH + "/purchase/:id", authorizedClient, (req, res) => {
    console.log(Date() + " - GET /purchase/" + req.params.id);


// POST a new purchase API method
app.post(BASE_API_PATH + "/purchase/", authorizedClient, async (req, res) => {
    console.log(Date() + " - POST /purchase/");


// PUT a specific purchase to change its state API method
app.put(BASE_API_PATH + "/purchase/:id", authorizedClient, async (req, res) => {
    console.log(Date() + " - PUT /purchase/" + req.params.id);


// DELETE a specific purchase API method
app.delete(BASE_API_PATH + "/purchase/:id", authorizedClient, async (req, res) => {
    console.log(Date() + " - DELETE /purchase/" + req.params.id);
```

En cada uno de los *endpoint*, se devuelve un código de estado dependiendo del resultado de la operación, por ejemplo, para el método *put* (se muestra este método por verse más claro y compacto):

```
Purchase.findOne({ _id: req.params.id }, async (err, purchase) => {
    if (err)
        return res.status(500).json("Internal server error");
    else if (purchase == null)
        return res.status(404).json("The purchase does not exist");
    else if (purchase.state != 'Pending')
        return res.status(400).json("The purchase is already accepted");
    else if (purchase.sellerId != req.id)
        return res.status(403).json("Unauthorized");

    purchase.state = 'Accepted';
    purchase.save(async (err) => {
        if (err)
            return res.status(500).json("Internal server error saving data to DB");
        else {
            console.log(Date() + " - Purchase accepted");
            await pubsub.publishMessage('updated-purchase', purchase);
            return res.status(200).json(await purchase.cleanedPurchase());
        }
    });
});
```

2.1.2. La API debe tener un mecanismo de autenticación.

Se utiliza un mecanismo de autenticación unificado para todos los microservicios y el *frontend*, basado en JWS. La definición de los métodos que se encargan de la autenticación en las llamadas a la API se encuentran en el archivo *authorised-roles.js*:

```
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",
    });
  }
};
```

Este método es referido en la definición de los *endpoint* de la API:

```
// GET purchases API method
app.get(BASE_API_PATH + "/purchase/", authorizedClient, (req, res) => {
    console.log(Date() + " - GET /purchase/");
```

Aparte de ello, tenemos guardada como variable de entorno el secret de JWT, como SECRET_KEY.

2.1.3. Debe tener un frontend que permita hacer todas las operaciones de la API (este frontend puede ser individual o estar integrado con el resto de frontends)

Este *frontend* común puede encontrarlo en su repositorio, y acceder al despliegue.

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

El despliegue del microservicio es https://api-fis-fersolesp.cloud.okteto.net (redirige automáticamente a la documentación en SWAGGER).

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

Hemos añadido un versionado a la dirección base de la API, como puede verse en el archivo *server.js*:

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

Por lo que para acceder al recurso, *purchase*, se accederá a (se necesita un *token* JWT válido):

https://api-fis-fersolesp.cloud.okteto.net/api/v1/purchase

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

La documentación la puede encontrar en:

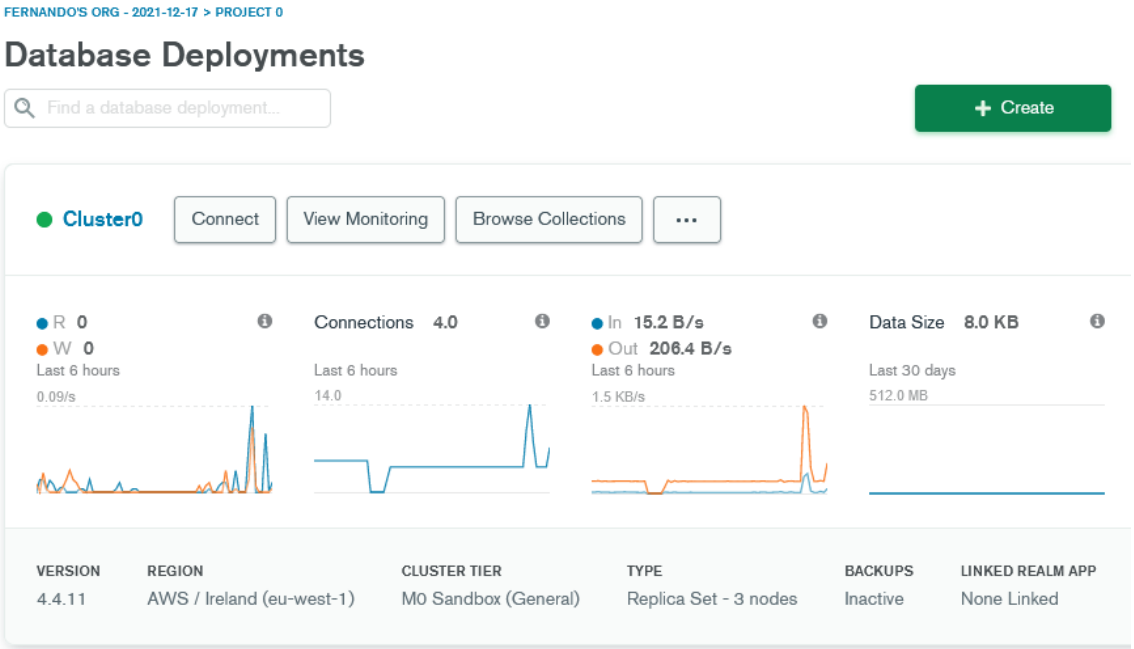
https://app.swaggerhub.com/apis-docs/sersanleo/Buy-service/1.0.0

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

Como puede comprobarse en *db.js*, se emplea una base de datos MongoAtlas cuyo acceso se define en la variable de entorno MONGO_URL:

```
1   const mongoose = require('mongoose');
2
3   const server = 'localhost:27017';
4   const database = 'buyService';
5
6   const DB_URL = (process.env.MONGO_URL || `mongodb://${server}/${database}`);
7
8   const dbConnect = function() {
9       const db = mongoose.connection;
10      db.on('error', console.error.bind(console, 'connection error: '));
11      return mongoose.connect(DB_URL, { useNewUrlParser: true });
12  }
13
14  module.exports = dbConnect;
```

Puede comprobar cómo está creada en MongoAtlas y funcionando:



2.1.8. Validación de los datos antes de almacenarlos en la base de datos (por ejemplo, haciendo uso de mongoose)

Se han definido las restricciones de validación en el modelo de la base de datos, *purchases.js*, a través de *moongose*:

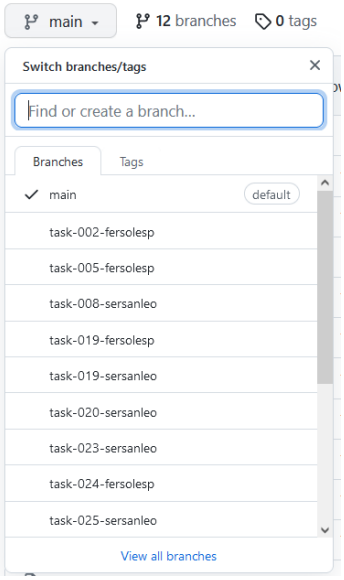
```
// We define the entity "Purchase" schema
const purchaseSchema = new mongoose.Schema({
    buyerId: {
        type: mongoose.Schema.Types.ObjectId,
        required: [true, "The purchase must have a buyer who created it"]
    },
    sellerId: {
        type: mongoose.Schema.Types.ObjectId,
        required: [true, "The purchase must have a seller who owns the asset"]
    },
    productId: {
        type: mongoose.Schema.Types.ObjectId,
        required: [true, "The purchase must have a product to be purchased"]
    },
    amount: {
        type: Number,
        min: [0, 'The purchase amount must be positive, got {VALUE}'],
        required: [true, "The purchase must have an asset to be purchased"]
    },
    state: {
        type: String,
        // In mongoose, enums are defined as strings and have a validator which checks if the
        // value is between one of the defined: https://stackoverflow.com/questions/29299477/how-to-create-and-use-enum-in-mongoose
        enum: {
            values: ['Pending', 'Accepted'],
            message: '{VALUE} is not a value in (Pending, Accepted)'
        },
        required: [true, "The purchase must have a state"]
    },
},
    {
        // This option assigns "createdAt" and "updatedAt" to the schema:
        // https://stackoverflow.com/questions/12669615/add-created-at-and-updated-at-fields-to-mongoose-schemas
        timestamps: true
    });
```

Posteriormente, en *server.js*, se realiza la validación llamando a la función *validateSync()*:

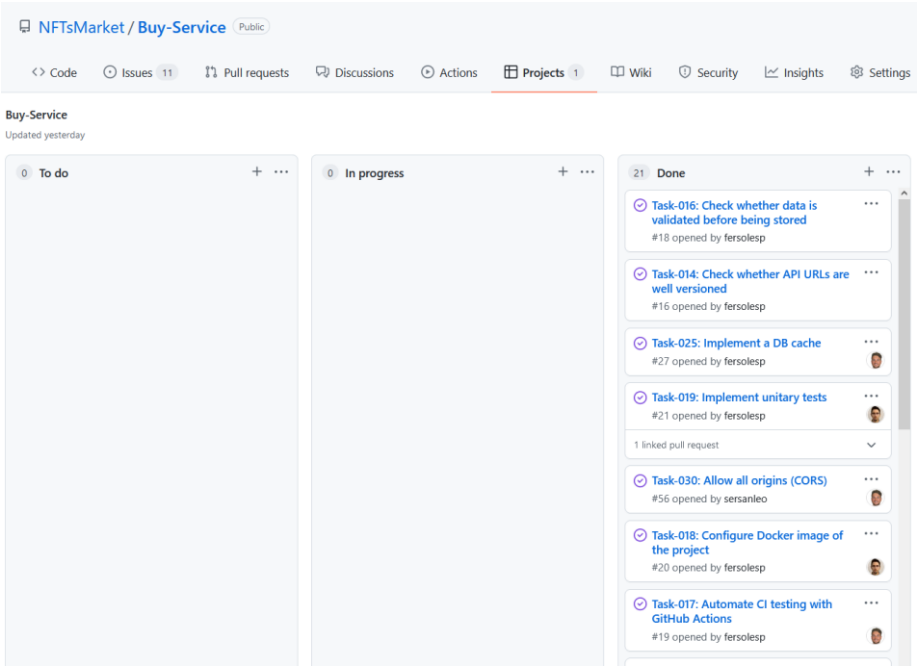
```
let validationErrors = purchase.validateSync();
if (validationErrors)
    return res.status(400).json(validationErrors.message);
else
```

2.1.9. Se debe utilizar gestión del código fuente y mecanismos de integración continua: El código debe estar subido a un repositorio de Github siguiendo Github flow. El código debe compilarse y probarse automáticamente usando GitHub Actions en cada commit

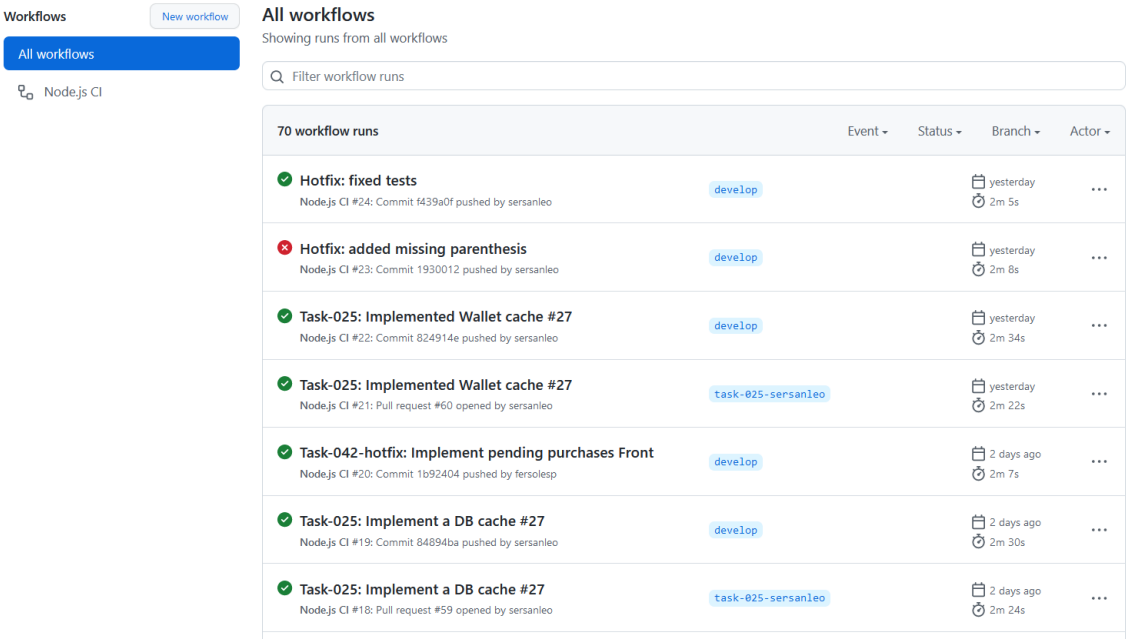
Puede ver cómo hemos seguido *GitHub Flow* y el manejo de ramas que indica:



También organizando las tareas en un tablero *Kamban* dentro del propio repositorio:

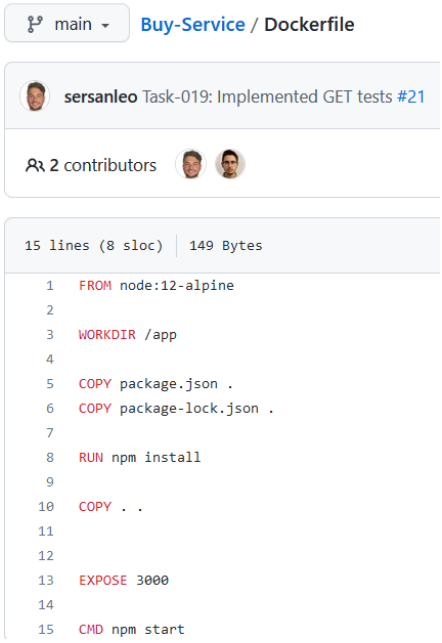


Por último, está configurada una *Action* que ejecuta las pruebas del microservicio y lo despliega en Okteto:



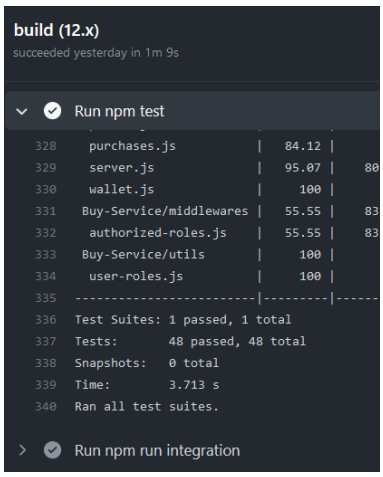
2.1.10. Debe haber definida una imagen Docker del proyecto

Se encuentra definida en el archivo Dockerfile del proyecto:



2.1.11. 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. Como norma general debe haber tests para todas las funciones no triviales de la aplicación. Probando tanto escenarios positivos como negativos

Se han implementado 48 tests unitarios, como puede comprobarse en el log de la ejecución de *Actions*:

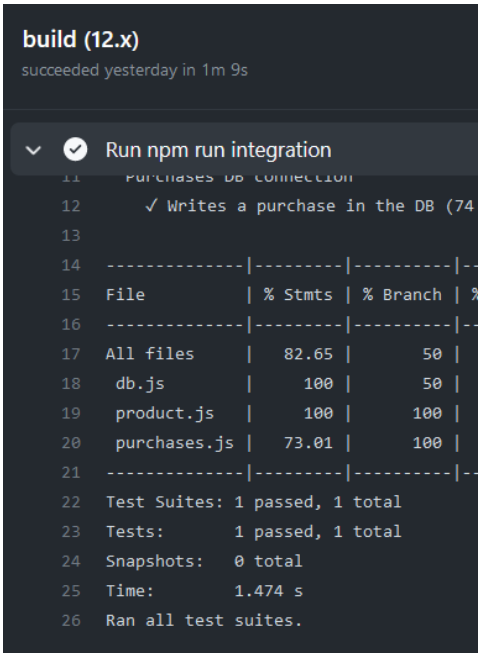


En cuanto a los test unitarios, podemos ver, por ejemplo, el del caso de éxito de *POST* en la base de datos, donde vemos cómo se *mockean* algunos métodos al principio, se realiza la llamada y se realizan las distintas comprobaciones posteriores:

```
it("should insert purchase in DB", () => {
    productFindOne.mockReturnValueOnce(product);
    purchaseExists.mockReturnValueOnce(false);
    walletFindOne.mockReturnValueOnce(richWallet);

    dbSave.mockImplementation((callback) => {
        callback(null);
    });

    let messageBody = {
        "g-recaptcha-response": "6LeIxAcTAAAAAJcZVRqyHh71UMIEGNQ_MXjiZKhI",
        "productId": "61e358d45a4dea9373b714db"
    };

    return request(app).post(BASE_API_PATH + "/purchase/").set("Authorization", 'Bearer ' + jwtToken).send(messageBody)
        .then((response) => {
            expect(response.statusCode).toBe(201);
            expect(response.body["productId"]).toEqual("61e358d45a4dea9373b714db");
            expect(Object.keys(response.body).length).toEqual(7);
            expect(pubsubPublishMessage).toHaveBeenCalledTimes(1);
        })
});
```

2.1.12. Debe haber pruebas de integración con la base de datos

Se ha realizado una prueba que verifica la correcta integración con la base de datos MongoAtlas:

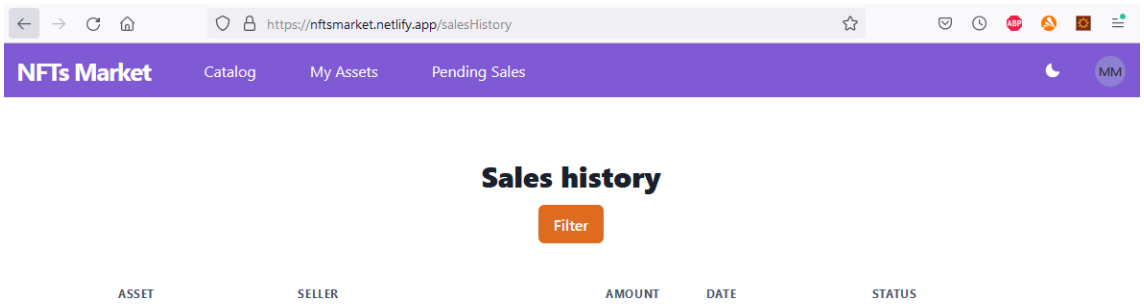


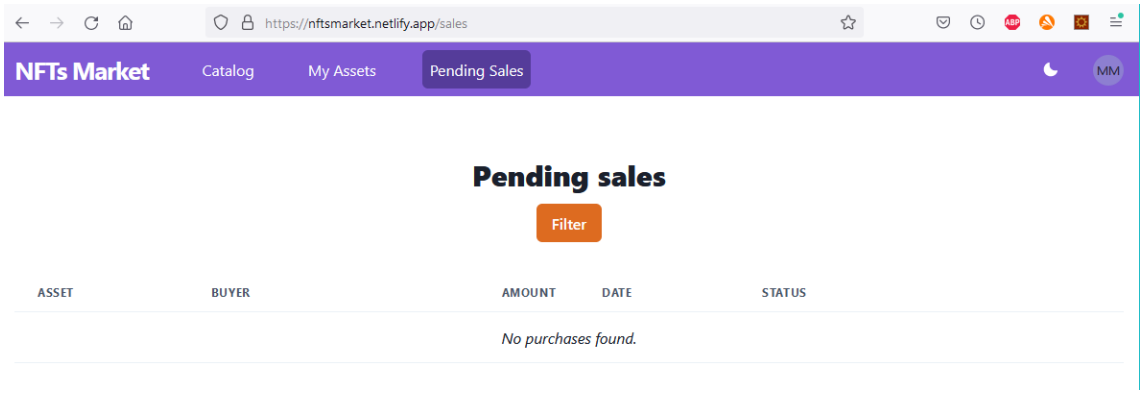

```
describe("Purchases DB connection", () => {
    // Connect to the database
    beforeAll(() => {
        return dbConnect();
    });

    //
    beforeEach((done) => {
        Purchase.deleteMany({}, (err) => {
            done();
        });
    });

    it("Writes a purchase in the DB", (done) => {
        const purchase = new Purchase({
            buyerId: "61bf5695b57c6e0471c7147b",
            sellerId: "61bf7d53888df2955682a7ea",
            productId: "61bf5695b57c6e0471c7147b",
            amount: 37,
            state: "Pending"
        });
        purchase.save((err, purchase) => {
            expect(err).toBeNull();
            Purchase.find({}, (err, purchases) => {
                expect(purchases).toHaveLength(1);
                done();
            });
        });
    });

    afterAll((done) => {
        mongoose.connection.db.dropDatabase(() => {
            mongoose.connection.close(done);
        });
    });
});
```

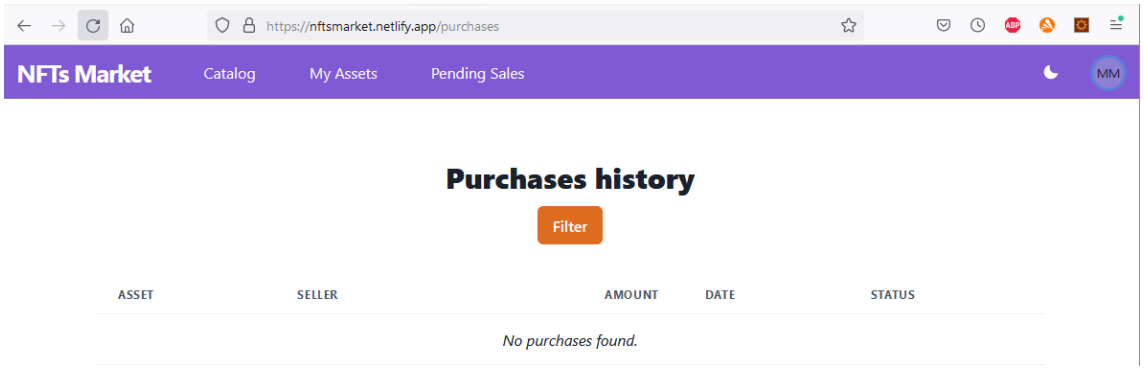
2.2. Microservicio avanzado que gestione un recurso

2.2.1. Implementar un frontend con rutas y navegación.

Puede comprobarse de forma práctica en el [despliegue del *frontend*]. Puede comprobar que existen rutas y la navegación es obvia con los botones de los menús:







2.2.2. Uso del patrón materialized view para mantener internamente el estado de otros microservicios.

Se ha implementado de forma conjunta con el mecanismo *pubsub*, de forma que nos encontramos suscritos a los cambios que se producen en los microservicios de *catalogue* y *wallet*, de los que guardamos la información que necesitamos de *product* y de *wallet*:

```
const productSchema = new mongoose.Schema({
    ownerId: {
        type: mongoose.Schema.Types.ObjectId,
        required: true
    },
    assetId: {
        type: mongoose.Schema.Types.ObjectId,
        required: true
    },
    price: {
        type: Number,
        min: 0,
        required: true
    }
});

const Product = mongoose.model('Product', productSchema);

module.exports = Product;
```

```
const walletSchema = new mongoose.Schema({
    userId: {
        type: mongoose.Schema.Types.ObjectId,
        required: true
    },
    funds: {
        type: Number,
        required: true
    }
});

const Wallet = mongoose.model('Wallet', walletSchema);

module.exports = Wallet;
```

En el archivo *pubsub.js* se implementan los eventos a los que nos suscribimos y manejan la información cacheada:

```
const subscribedTopics = [
  'created-product',
  'updated-product',
  'deleted-product',
  'created-wallet',
  'updated-wallet',
  'deleted-wallet'
];
```

Por ejemplo, cuando se reciba que ha sido creado un nuevo producto en el microservicio de *catalogue*, almacenaremos la información que nos interesa del mismo:

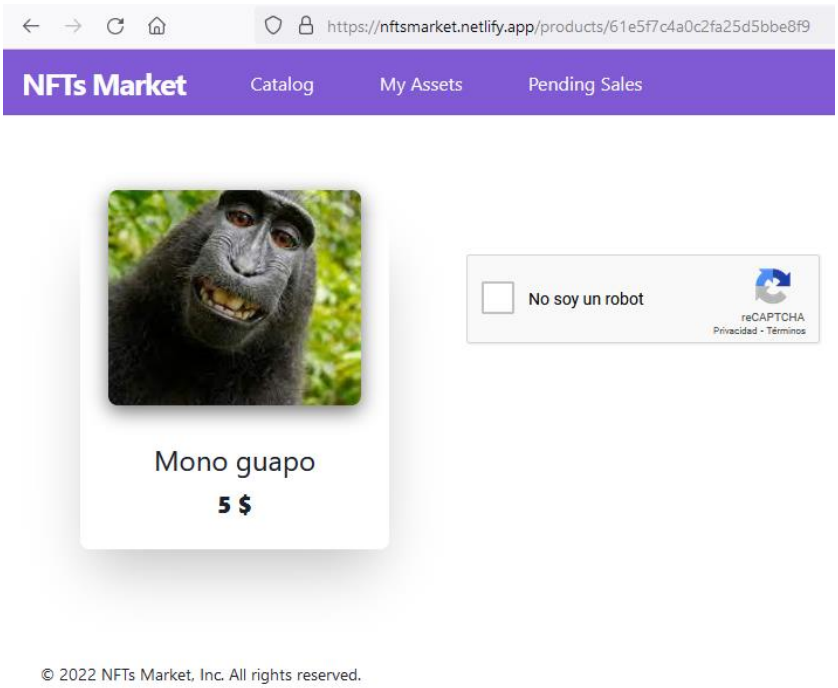
```
subscriptions['created-product'].on('message', async message => {
  let product = JSON.parse(message.data);

  try {
    await Product.create({
      _id: product.id,
      ownerId: product.owner,
      assetId: product.picture,
      price: product.price
    });
  } catch { }
});
```

Por último, en *server.js*, en la funcionalidad de nuestros *endpoint*, hacemos uso de esa información almacenada:

```
const product = await Product.findOne({ _id: productId });
if (!product)
    return res.status(404).json("Product not found");
else if (req.id == product.ownerId)
    return res.status(403).json("You can not buy your own product");
```

2.2.3. Consumo de algún API externa (distinta de las de los grupos de práctica).

Hemos implementado la API de ReCaptcha de Google, que verifica que el usuario que va a realizar la compra de un NFT es humano:



En la parte de servidor, en *server.js*, verificamos que el *captha* es válido y continuamos con la operación de POST de la nueva compra:

```
if (req.body['g-recaptcha-response'] === undefined || req.body['g-recaptcha-response'] === '' || req.body['g-recaptcha-response'] === null)
    return res.status(400).json('Missing reCAPTCHA response.');
else {
    let verificationUrl = "https://www.google.com/recaptcha/api/siteverify?secret=" + process.env.RECAPTCHA_SECRET_KEY + "&response=" + req.body['g-recaptcha-res

    request(verificationUrl, { json: true }, (error, response, body) => {
        // Success will be true or false depending upon captcha validation.
        if (body.success !== undefined && !body.success)
            return res.status(401).json('Invalid reCAPTCHA response.');
        else {
            let purchase = new Purchase({
                buyerId: req.id,
                sellerId: product.ownerId,
                productId: product._id,
                amount: product.price,
                state: 'Pending',
            });
```

En el *frontend*, lo encontramos en el archivo *BuyProduct.jsx*:

```
<ReCAPTCHA
  ref={recaptchaRef}
  sitekey="6Leg7agdAAAAAKZqXGYJkFDoaSLEd6VnikUIefgE"
  onChange={onCaptcha}
/>
```

2.2.4. Tener el API REST documentado con swagger.

Puede encontrar la documentación en SWAGGER en:

https://app.swaggerhub.com/apis-docs/sersanleo/Buy-service/1.0.0

2.2.5. Implementación de un mecanismo de autenticación basado en JWT o equivalente.

Se ha implementado JWT de forma homogénea en todos los microservicios y *frontend*, explicado en el punto 2.1.2 de este documento.

2.3. Aplicación basada en microservicios básica

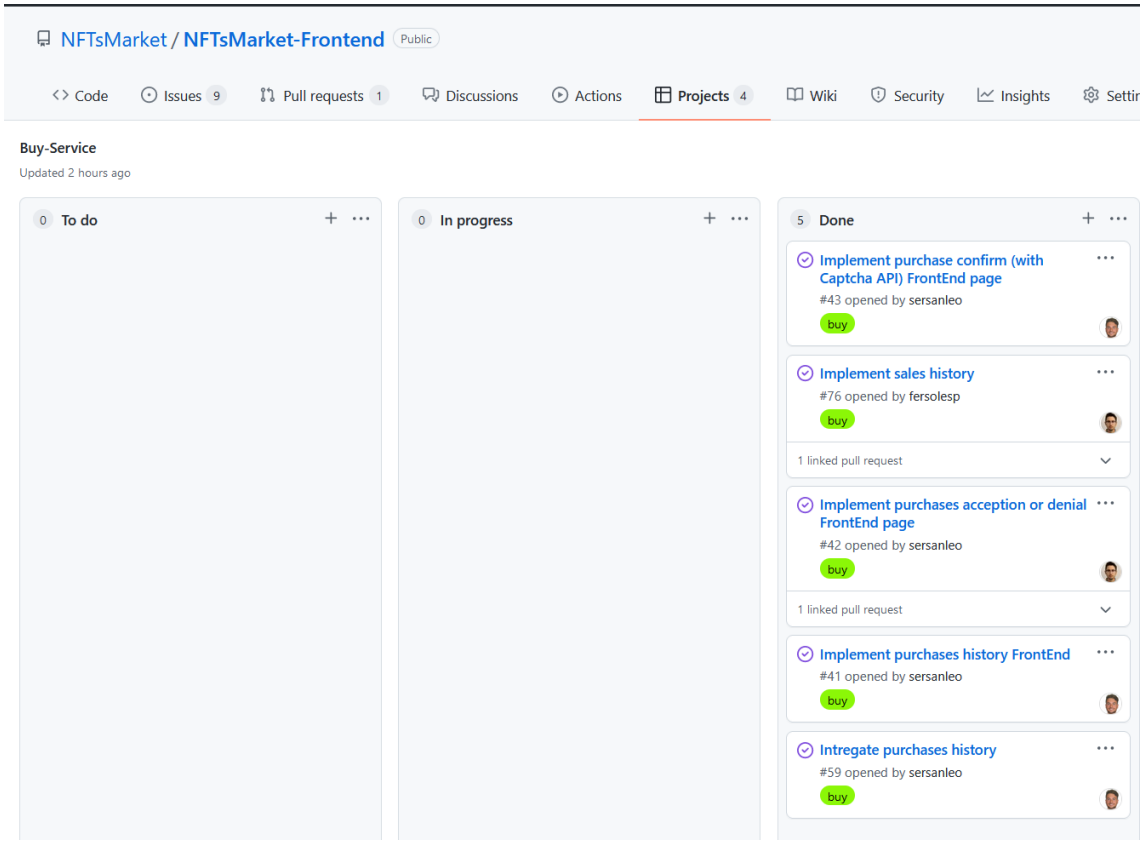
2.3.1. Interacción completa entre todos los microservicios de la aplicación integrando información

Puede comprobar esta integración accediendo al despliegue del *frontend* y utilizando la aplicación en su conjunto.

2.4. Aplicación basada en microservicios avanzada

2.4.1. 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.

Puede acceder al despliegue del *frontend* o a su repositorio, en el cual podrá encontrar el tablero con las tareas de la implementación de nuestra parte de *frontend*:



Hemos implementado los archivos que se encuentran dentro de la carpeta *components/buy* del proyecto del *frontend*, *BuyApi.js* y *BuyProduct.jsx*, donde se realiza la conexión y la interacción con el API *Gateway* que conecta, a su vez, con nuestra API; y donde se implementa el botón y el *captcha* para realizar la compra, respectivamente.

También, hemos implementado los archivos *pages/purchases.jsx*, *pages/sales.jsx* y *pages/salesHistory.jsx*, que permiten mostrar el historial de compras del usuario, las ventas que tiene pendientes aceptar o rechazar y el historial de las ventas realizadas, respectivamente.

2.4.2. Hacer uso de un API Gateway con funcionalidad avanzada como un mecanismo de throttling o de autenticación.

Usamos una API *Gateway* que hace de punto de entrada a las API de todos los microservicios y cuya implementación puede encontrar en:

https://github.com/NFTsMarket/NFTs-Market-Api-Microservices/tree/main/apps/api-gateway

2.4.3. Definición de un customer agreement para la aplicación en su conjunto.

Se ha definido un acuerdo a nivel de servicio global para la aplicación.

2.4.4. 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 detalla en el punto 2.2.2, se ha implementado un mecanismo de *pubsub*, o publicación - subscripción entre los microservicios, de forma que el nuestro notifica a los demás cuando se crea/modifica/elimina una compra para que puedan actualizarse la propiedad del producto y la cantidad de dinero de la cartera del usuario, y recibimos los eventos de producto y *wallet*, para almacenarlos en nuestro servicio (*materialized view*). Toda esta implementación puede verse en el archivo *pubsub.js*.

2.4.5. Implementación de un mecanismo de autenticación homogéneo para todos los microservicios.

De nuevo, se ha implementado JWT de forma homogénea en todos los microservicios y *frontend*, explicado en el punto 2.1.2 de este documento.

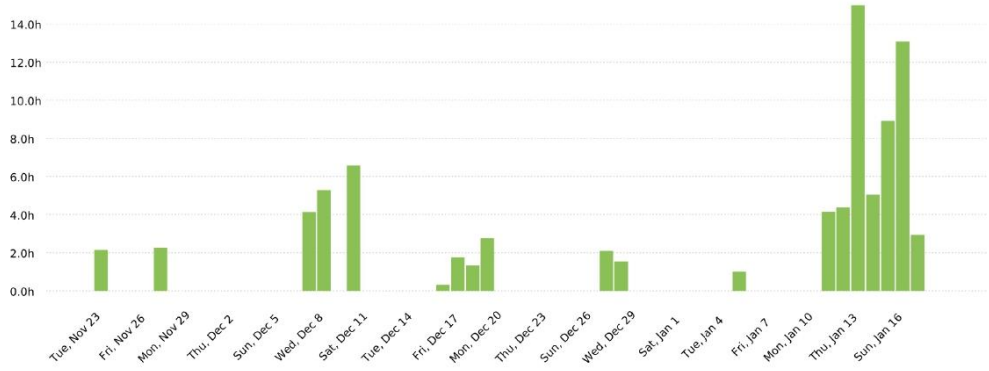
3. Análisis de los esfuerzos

Summary report

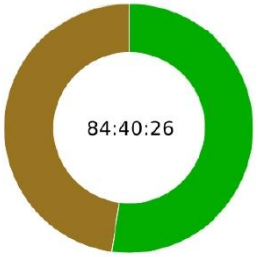
Clockify

11/23/2021 - 01/18/2022

Total: 84:40:26 Billable: 84:40:26 Amount: 0.00 USD

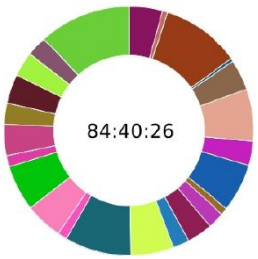


User



Fersolesp	40:29:59	47.83%
Sergio	44:10:27	52.16%

Description



Task-019: Implement unitary tests	10:07:04	11.95%
Reunión definición API y prueba de concepto	02:08:43	2.54%
Front	02:44:59	3.25%
Task-020: Implement integration tests with DB #22	03:12:14	3.78%
Mockups	02:15:00	2.66%

FIS Created with Clockify 1

Task-010: Implement purchases history FrontEnd	03:28:06	4.10%
Task-003: Implement POST a new purchase - reCaptcha	01:10:32	1.39%
Implement sales history	04:58:36	5.88%
Proyecto base	04:07:52	4.88%
Task-008: Implement GET the history of a user purchases	01:01:04	1.20%
(Without Description)	07:17:23	8.61%
Task-013: Automate Okteto deployment of BackEnd	04:41:43	5.55%
Task-023: API documentation with SWAGGER	01:47:41	2.11%
Implement purchases acception or denial FrontEnd page	02:48:24	3.32%
Task-024: Implement authentication with JWT	01:48:16	2.13%
Task-005: Implement PUT a purchase to change its state	00:48:47	0.96%
Task-002: Implement GET every purchase	05:16:52	6.24%
Task-003: Implement POST a new purchase	02:37:41	3.11%
Intregate purchases history #59	05:40:16	6.70%
Task-019: Implement unitary tests #21	03:24:37	4.03%
Task-006: Implement DELETE a purchase	00:18:31	0.37%
Task-029: Pub/Sub implementation #41	08:45:58	10.36%
Task-004: Implement GET a specific purchase	00:30:25	0.60%
Documentación de la implementación del microservicio	03:39:42	4.32%

User / Description	Duration	Amount
Fersolesp	**40:29:59**	**0.00 USD**
Task-019: Implement unitary tests	10:07:04	0.00 USD
Reunión definición API y prueba de concepto	01:04:43	0.00 USD
Mockups	01:08:00	0.00 USD
Implement sales history	04:58:36	0.00 USD
Proyecto base	04:07:52	0.00 USD

Task-013: Automate Okteto deployment of BackEnd	04:41:43	0.00 USD
Implement purchases acception or denial FrontEnd page	02:48:24	0.00 USD
Task-024: Implement authentication with JWT	01:48:16	0.00 USD
Task-005: Implement PUT a purchase to change its state	00:48:47	0.00 USD
Task-002: Implement GET every purchase	05:16:52	0.00 USD
Documentación de la implementación del microservicio	03:39:42	0.00 USD
Sergio	**44:10:27**	**0.00 USD**
Front	02:44:59	0.00 USD
Task-020: Implement integration tests with DB #22	03:12:14	0.00 USD
Task-010: Implement purchases history FrontEnd	03:28:06	0.00 USD
Task-003: Implement POST a new purchase - reCaptcha	01:10:32	0.00 USD
Task-008: Implement GET the history of a user purchases	01:01:04	0.00 USD
Mockups	01:07:00	0.00 USD
(Without Description)	07:17:23	0.00 USD
Reunión definición API y prueba de concepto	01:04:00	0.00 USD
Task-023: API documentation with SWAGGER	01:47:41	0.00 USD
Task-003: Implement POST a new purchase	02:37:41	0.00 USD
Intregate purchases history #59	05:40:16	0.00 USD
Task-019: Implement unitary tests #21	03:24:37	0.00 USD
Task-006: Implement DELETE a purchase	00:18:31	0.00 USD
Task-029: Pub/Sub implementation #41	08:45:58	0.00 USD
Task-004: Implement GET a specific purchase	00:30:25	0.00 USD