

PROYECTO FINAL ADMINISTRACIÓN DE SISTEMAS INFORMÁTICOS EN RED

**DESARROLLO Y DESPLIEGUE DE
UNA APLICACIÓN WEB BASADA
EN MICROSERVICIOS CON
KUBERNETES**

2023

Índice

CAPÍTULO 1: INTRODUCCIÓN.....	6
1.1-Introducción del proyecto.....	7
1.2-Propósito.....	7
1.2.1-Supuesto del proyecto.....	7
CAPÍTULO 2: ANÁLISIS DEL SISTEMA.....	9
2.1-Análisis de requisitos.....	10
2.1.1-Requerimientos de Hardware.....	10
2.1.2-Requerimientos de Software.....	10
2.1.2.1-Programas para el desarrollo.....	10
2.1.2.2-Programas para el despliegue en local.....	10
CAPÍTULO 3: DOCUMENTACIÓN DEL PROGRAMA.....	11
3.1-Introducción.....	12
3.2-Estructura de las bases de datos.....	12
3.2.1-Base de datos MongoDB.....	12
3.2.2-Base de datos MySQL.....	13
3.3-Documentación del código.....	14
3.3.1-Servidor GraphQL.....	14
3.3.1.1-Fichero typeDefs.....	16
3.3.1.2-Fichero resolvers.....	18
3.3.1.3-Funciones del fichero queries.....	19
3.3.2-Frontend.....	22
3.3.2.1-Componente Home.....	23
3.3.2.2-Componente Products.....	24
3.3.2.3-Componente ProductOne.....	26
3.3.2.4-Componente Register.....	27
3.3.2.5-Componente Cart.....	29
3.3.2.6-Componente Compras.....	30
3.3.3-Microservicio de Productos.....	31
3.3.3.1-Controlador de Productos.....	31
3.3.3.2-Controlador de Buscador.....	32
3.3.3.3-Campo estrellas.....	34
3.3.4-Microservicio de Usuarios.....	35
3.3.4.1-Controlador de users.....	35
3.3.4.2-Controlador de login.....	38
3.3.5-Microservicio de Compras.....	40
3.3.5.1-Controlador de compras.....	40
3.4-Documentación del desarrollo del despliegue.....	43
3.4.1-Creación de las imágenes con Docker.....	43
3.4.1.1-Creación de la imagen del Frontend.....	43
3.4.1.2-Creación de la imagen del servidor GraphQL.....	44
3.4.1.3-Creación de la imagen del microservicio de Productos.....	45
3.4.1.4-Creación de la imagen del microservicio de Usuarios.....	45
3.4.1.5-Creación de la imagen del microservicio de Compras.....	46
3.4.1.6-Creación de la imagen de la base de datos MySQL.....	48
3.4.1.7-Creación de la imagen de la base de datos MongoDB.....	48
3.4.1.8-Imágenes en Docker Hub.....	49

3.4.2-Creación de los archivos para el despliegue en Kubernetes.....	50
3.4.2.1-Deployment y Service de nuestro Frontend.....	51
3.4.2.2-Deployment y Service de nuestro servidor GraphQL.....	52
3.4.2.3-Deployment y Service de nuestro microservicio de Productos.....	53
3.4.2.4-Deployment y Service de nuestro microservicio de Usuarios.....	54
3.4.2.5-Deployment y Service de nuestro microservicio de Compras.....	55
3.4.2.6-Deployment, Service y Volumen de MongoDB.....	55
3.4.2.7-Deployment, Service y Volumen de MySQL.....	57
CAPÍTULO 4: PRUEBAS DE LA APLICACIÓN.....	60
4.1-Introducción.....	61
4.2-Realizando pruebas al microservicio de productos.....	61
4.2.1-Test para pedir todos los productos.....	61
4.2.2-Test para pedir tres productos.....	61
4.2.4-Test del buscador correcto.....	62
4.2.5-Test del buscador erróneo.....	62
4.2.6-Test productos con descuento.....	62
4.2.7-Test filtrar por precio correcto.....	63
4.2.8-Test filtrar por precio erróneo.....	63
4.2.9-Test buscar un producto correcto.....	63
4.2.10-Test buscar un producto erróneo.....	64
4.2.11-Test añadir una valoración.....	64
4.2.12-Test añadir una valoración errónea.....	65
4.2.13-Ejecución de los tests.....	65
4.3-Realizando pruebas al microservicio de usuarios.....	66
4.3.1-Test login correcto.....	66
4.3.2-Test login con contraseña incorrecta.....	66
4.3.3-Test login con usuario incorrecto.....	66
4.3.4-Test creación de usuario correcta.....	67
4.3.5-Test creación de usuario con username erróneo.....	67
4.3.6-Test creación de usuario con contraseña errónea.....	67
4.3.7-Test creación de usuario con número de cuenta erróneo.....	68
4.3.8-Test creación de usuario con username repetido.....	68
4.3.9-Test creación de usuario con email repetido.....	69
4.3.10-Test añadir dinero al monedero.....	69
4.3.11-Test añadir dinero al monedero de un usuario inexistente.....	70
4.3.12-Test añadir dinero por encima del límite.....	70
4.3.13-Ejecución de los test.....	70
4.4-Realizando pruebas al microservicio de compras.....	71
4.4.1-Test realizar una compra.....	71
4.4.2-Test realizar una compra sin mandar token.....	71
4.4.3-Test realizar una compra con usuario inexistente.....	72
4.4.4-Test conseguir todas las compras del usuario 1.....	73
4.4.5-Test conseguir compras sin mandar token.....	73
4.4.6-Test conseguir compras de un usuario inexistente.....	73
4.4.7-Ejecución de los test.....	74
4.5-Realizando pruebas al servidor GraphQL.....	75
4.5.1-Test microservicio de usuarios.....	75
4.5.2-Test errores del microservicio de usuarios.....	75

4.5.3-Test microservicio de productos.....	76
4.5.4-Test errores del microservicio de productos.....	76
4.5.5-Test microservicio de compras.....	76
4.5.6-Test errores del microservicio de compras.....	77
4.5.7-Ejecución de los test.....	77
4.6-Realizando pruebas e2e desde el Frontend.....	78
4.6.1-Testing creación de usuarios.....	78
4.6.1.1-Comprobación del menú nologin.....	78
4.6.1.2-Comprobación del formulario de registro.....	78
4.6.1.3-Comprobación error “campos obligatorios” formulario de registro.....	79
4.6.1.4-Comprobación error “contraseñas diferentes” formulario de registro.....	79
4.6.1.5-Comprobación error “usuario en uso” formulario de registro.....	79
4.6.1.6-Comprobación creación de usuario correcta.....	80
4.6.1.7-Ejecución de los test.....	80
4.6.2-Testing del login de usuarios.....	80
4.6.2.1-Comprobación error “contraseña incorrecta” login.....	81
4.6.2.2-Comprobación error “usuario incorrecto” login.....	81
4.6.2.3-Comprobación login correcto.....	81
4.6.2.4-Comprobación deslogueo correcto.....	82
4.6.2.5-Comprobación eliminar usuario.....	82
4.6.2.6-Ejecución de los test.....	82
4.6.3-Testing del los productos.....	83
4.6.3.1-Comprobación error “no registrado” realizar valoración.....	83
4.6.3.2-Comprobación realizar valoración.....	83
4.6.3.3-Comprobación añadir producto al carrito.....	83
4.6.3.4-Comprobación comprar el producto.....	84
4.6.3.5-Comprobación ver estado del envío.....	84
4.6.3.6-Ejecución de los test.....	85
4.6.4-Testing de navegación.....	85
4.6.4.1-Comprobación del home.....	85
4.6.4.2-Comprobación del apartado de productos.....	86
4.6.4.3-Comprobación de los filtros.....	86
4.6.4.4-Comprobación del buscador.....	86
4.6.4.5-Comprobación único producto.....	87
4.6.4.6-Comprobación error “no logueado” al comprar.....	87
4.6.4.7-Ejecución de los test.....	87
CAPÍTULO 5: CONCLUSIONES.....	88
5.1-Conclusiones sobre el proyecto.....	89
5.2-Conclusiones finales.....	89
CAPÍTULO 6: BIBLIOGRAFÍA Y REFERENCIAS.....	90
6.1-Contenido para el desarrollo de la aplicación.....	91
6.2-Contenido para el despliegue de la aplicación.....	91
ANEXO 1: MANUAL DE INSTALACIÓN.....	92
1.1-Introducción.....	93
1.2-Instalación del proyecto con Docker.....	93
1.3-Instalación del proyecto con Kubernetes.....	96
1.4-Instalación del proyecto en Google Cloud.....	98
ANEXO 2: MANUAL DE USUARIO.....	102

2.1-Introducción.....	103
2.2-Registro y acceso.....	103
2.2.1-Creación de una cuenta.....	103
2.2.2-Iniciar sesión con una cuenta.....	105
2.2.3-Cerrar sesión.....	106
2.2.4-Borrar usuario.....	107
2.3-Explorar productos.....	109
2.3.1-Pantallas con productos.....	109
2.3.2-Búsqueda de productos.....	111
2.3.3-Filtros y opciones de ordenamiento.....	111
2.4-Detalle del producto.....	113
2.4.1-Información del producto.....	113
2.4.2-Valoración del producto.....	114
2.5-Funciones del carrito y compra del producto.....	116
2.5.1-Agregar productos al carrito.....	116
2.5.2-Proceso de compra.....	118
2.5.3-Gestión de pedidos.....	120

CAPÍTULO 1: INTRODUCCIÓN

1.1-Introducción del proyecto.

Las ideas principales que tuve para realizar este proyecto fueron varias:

- Mostrar todo el ciclo de vida de una aplicación, desde la planificación, el desarrollo, la realización de pruebas y el despliegue hasta la puesta en producción.
- Presentar las ventajas de usar una arquitectura basada en microservicios y aprovecharla para la reutilización de código.
- Realizar el despliegue de la aplicación con Kubernetes, una tecnología que tenía muchas ganas de aprender y es fantástica para desplegar microservicios.
- Trabajar un poco de cada asignatura como SGBD (usando bases de datos como MySQL y MongoDB), Servicios de red e internet (desplegando la aplicación con Apache), Implementación de aplicaciones web (desarrollando una aplicación web compleja y funcional), Administración de sistemas operativos (todos los contenedores son Linux y he desarrollado scripts para la automatización de los despliegues), entre otras.

1.2-Propósito.

La aplicación es una tienda Online totalmente funcional donde podrás ver, buscar y comprar artículos. Cuenta con carrito de la compra, sistema de valoraciones de los productos, filtros de búsqueda, entre otras funcionalidades.

Consta de 7 contenedores, un Frontend desarrollado con React.js y Typescript, tres microservicios (uno para gestión de usuarios desarrollado con Python y Django, otro para gestión de productos desarrollado con JavaScript corriendo en Node.js, y el último desarrollado en PHP con Laravel), un servidor GraphQL que actuá de intermediario comunicando el Frontend con los microservicios y dos bases de datos, una MongoDB donde guardamos los productos y otra MySQL donde se aloja la información respectiva a la aplicación.

1.2.1-Supuesto del proyecto.

Se elige este escenario en simulación de una empresa que desarrolla sus propios productos digitales, para ello usa distintos lenguajes de programación y tecnologías. En este supuesto la empresa cuenta con un servicio de gestión de usuarios desarrollado con Python, que es el que usan globalmente en todos sus productos. La empresa también cuenta con un servicio de registro de ventas desarrollado en PHP, ya que no es el primer proyecto en el que venden algún producto.

La nueva aplicación web desean desarrollarla usando Node.js tanto para Frontend como para Backend y alojar la información de los productos en una base de datos MongoDB, pero piensan que, para ahorrar trabajo y reutilizar código que ya está en producción, aprovecharán el sistema de ventas anteriormente implementado, y como en todos sus proyectos, la gestión de usuarios se realizará desde su propio microservicio común.

Esta empresa tiene pensado realizar el despliegue de la aplicación en sus servidores Cloud usando Kubernetes y creando sus propias imágenes de Docker.

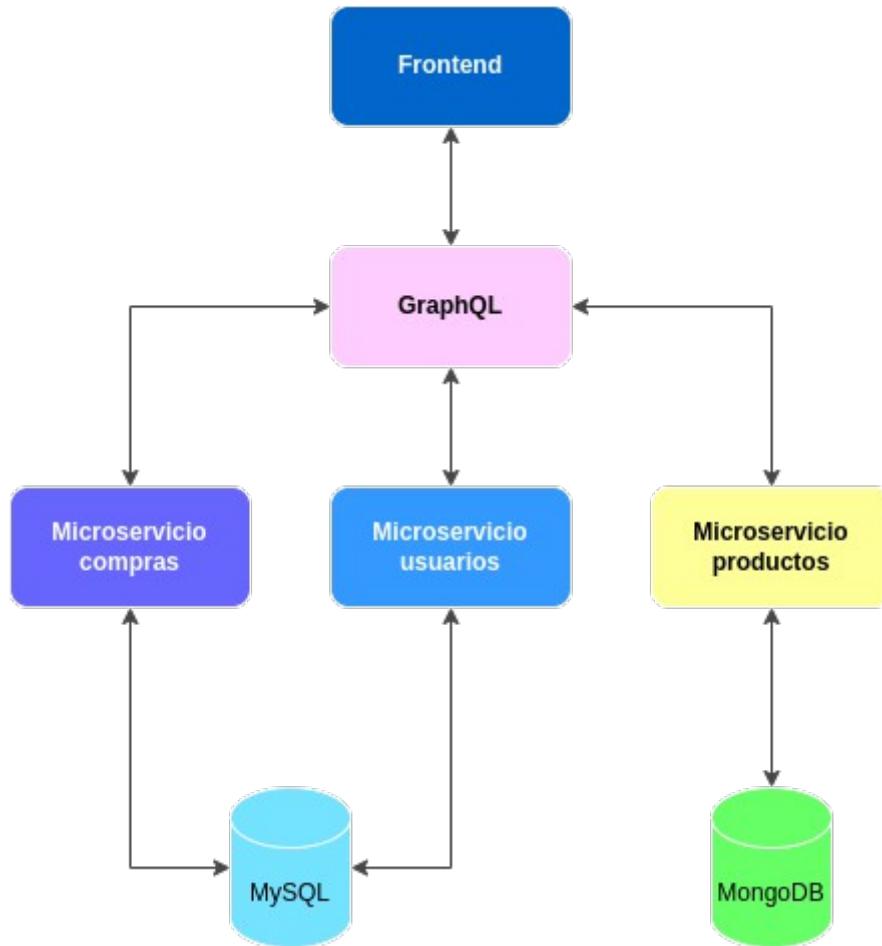


Diagrama que muestra la composición de la aplicación.

Con este escenario veremos como la forma en la que un microservicio está desarrollado es transparente para el resto, independientemente del lenguaje de programación o la base de datos que utilice. Veremos como aplicaciones que individualmente son tan diferentes entre si, pueden coexistir en perfecta armonía y funcionar perfectamente.

CAPÍTULO 2: ANÁLISIS DEL SISTEMA

2.1-Análisis de requisitos.

En este capítulo analizaremos los requisitos que he necesitado para realizar el proyecto según los puntos planteados anteriormente.

2.1.1-Requerimientos de Hardware.

Para realizar este proyecto requerimos de un ordenador donde poder desarrollarlo en local y de varios servidores que desplegarán el proyecto en producción orquestados por Kubernetes.

Los servidores los he conseguido gracias a Google Cloud System y a su prueba gratuita, permitiéndome poder desplegar el proyecto en la nube.

2.1.2-Requerimientos de Software.

Para el desarrollo del proyecto he necesitado de varios programas tanto para el desarrollo como para el despliegue. Los programas son los siguientes:

2.1.2.1-Programas para el desarrollo.

Para el desarrollo de la aplicación en su totalidad requerimos de varias herramientas:

- La primera es un IDE para poder escribir nuestro código cómodamente, en mi caso he usado Visual Studio Code.
- Obviamente un navegador con el que poder acceder a nuestra página web.
- Necesitamos tener instalados los entornos de Node.js, Python y PHP para poder desarrollar y ejecutar cada uno de los microservicios.
- MySQL y MongoDB, yo he preferido optar por correrlos en un contenedor de Docker ya que a la hora del desarrollo son bases de datos de prueba y así no tengo la necesidad de instalarlos en mi máquina anfitriona.
- Opcionalmente podemos instalar un programa como Postman, que nos permite probar los microservicios independientemente lanzando contra ellos una petición HTTP REST.

2.1.2.2-Programas para el despliegue en local.

Para las pruebas de despliegue en local de la aplicación necesitamos tener DockerDesktop instalado. Esta herramienta nos permitirá generar las imágenes de Docker para su posterior despliegue con Kubernetes, desplegar una versión de prueba para comprobar que las imágenes funcionan correctamente a través de un archivo docker-compose.yml y viene con una versión de Kubernetes instalada, por lo que con ella podremos realizar el despliegue local de la aplicación en su totalidad y pasarla a Google Cloud cuando funcione correctamente.

CAPÍTULO 3: DOCUMENTACIÓN DEL PROGRAMA

3.1-Introducción.

En este capítulo veremos de forma detallada como esta construido el proyecto y los distintos componentes que lo forman, también veremos como se ha realizado el despliegue del mismo y explicaremos tanto el código como los fichero para los despliegos.

3.2-Estructura de las bases de datos.

Como ya hemos comentado varias veces, esta aplicación se comunica con dos bases de datos, una no relacional MongoDB y una relacional MySQL. En el siguiente punto veremos su estructura y las tablas/colecciones que las conforman.

3.2.1-Base de datos MongoDB.

Es la base de datos que se diseña para guardar los productos que están disponibles en la tienda. Cuenta con una única colección llamada “products” donde sus documentos tienen la siguiente estructura:

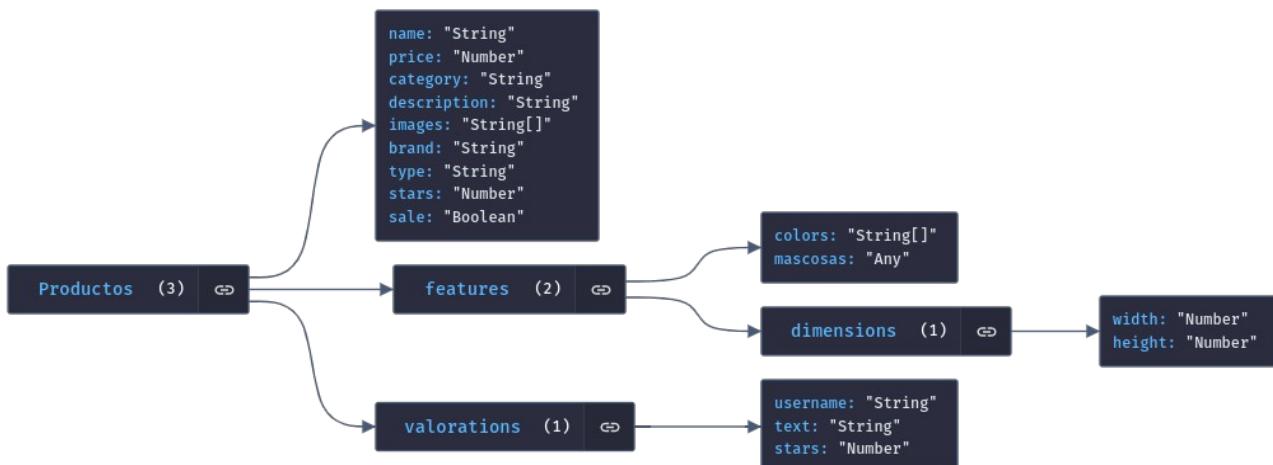


Diagrama de la base de datos MongoDB.

- Varios campos de tipos primitivos como el nombre, la marca, el tipo o el precio.
- Un campo images donde guardamos un array con la ruta de las imágenes del producto.
- Un campo features donde guardamos las características del producto, este campo varía mucho de un artículo a otro y puedes añadir tantas características como deseas. Este campo obligatoriamente debe tener los campos colors, que es un array de los colores del producto, y dimensions que a su vez está compuesto por una altura y un ancho.
- Un campo valorations donde guardamos las valoraciones que han realizado los usuarios sobre el producto, cuenta con los campos username, text y stars.
- Un campo principal stars donde se guarda la media de estrellas que tiene un producto. Este campo no se encuentra implícito en la base de datos, sino que es MongoDB el que lo calcula y lo añade a los documentos.

3.2.2-Base de datos MySQL.

Esta base de datos es común para los microservicios de compra y usuarios, cuenta con cuatro tablas y sus respectivos campos. Una tabla de usuarios, una tabla monedero que es entidad débil de usuario, una tabla compra que también es entidad débil de usuarios y una tabla articulo que es entidad débil de compra.

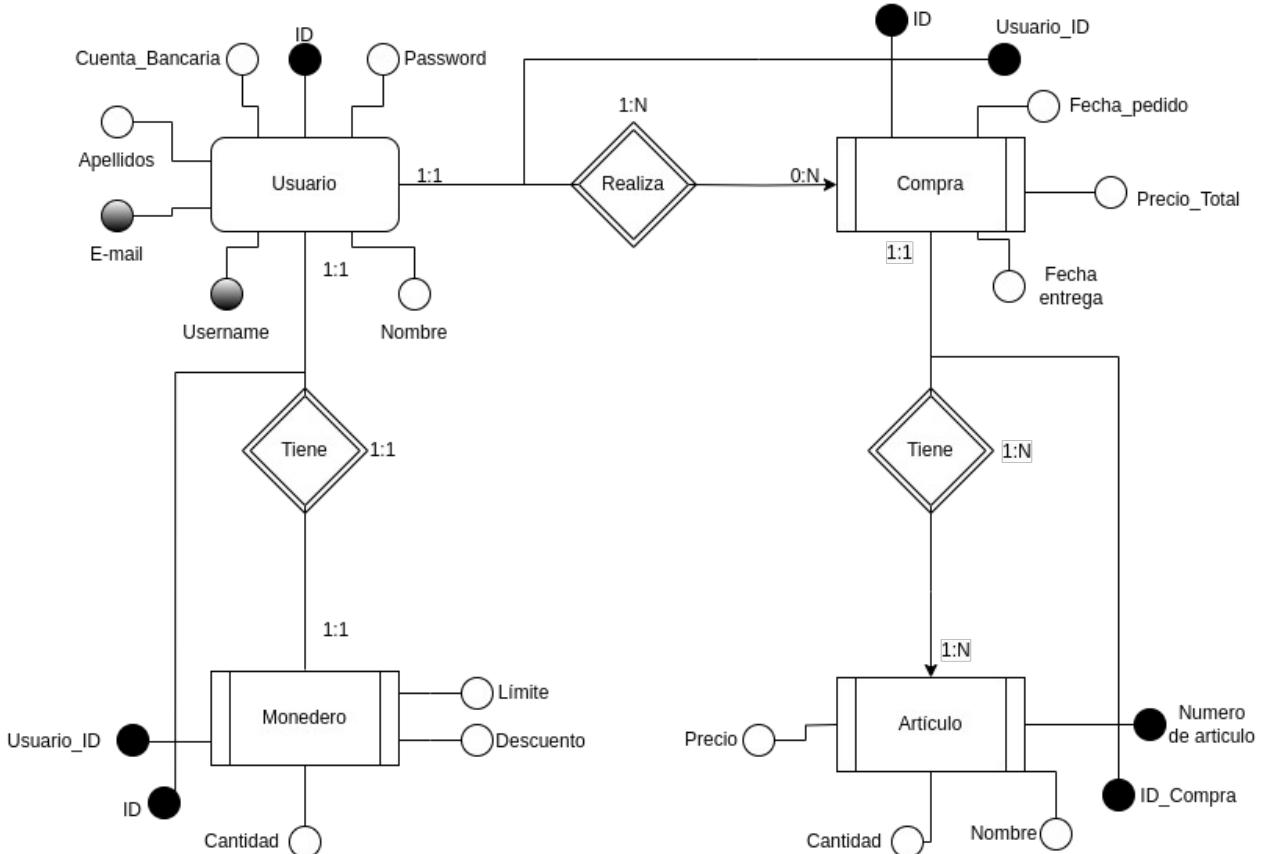


Diagrama ER de la base de datos.

Así quedaría su diagrama relacional:

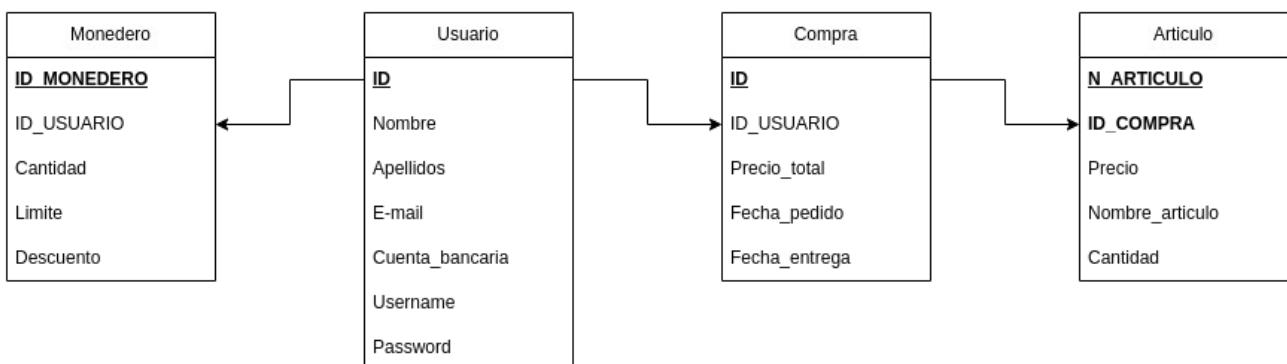


Diagrama relacional de la base de datos

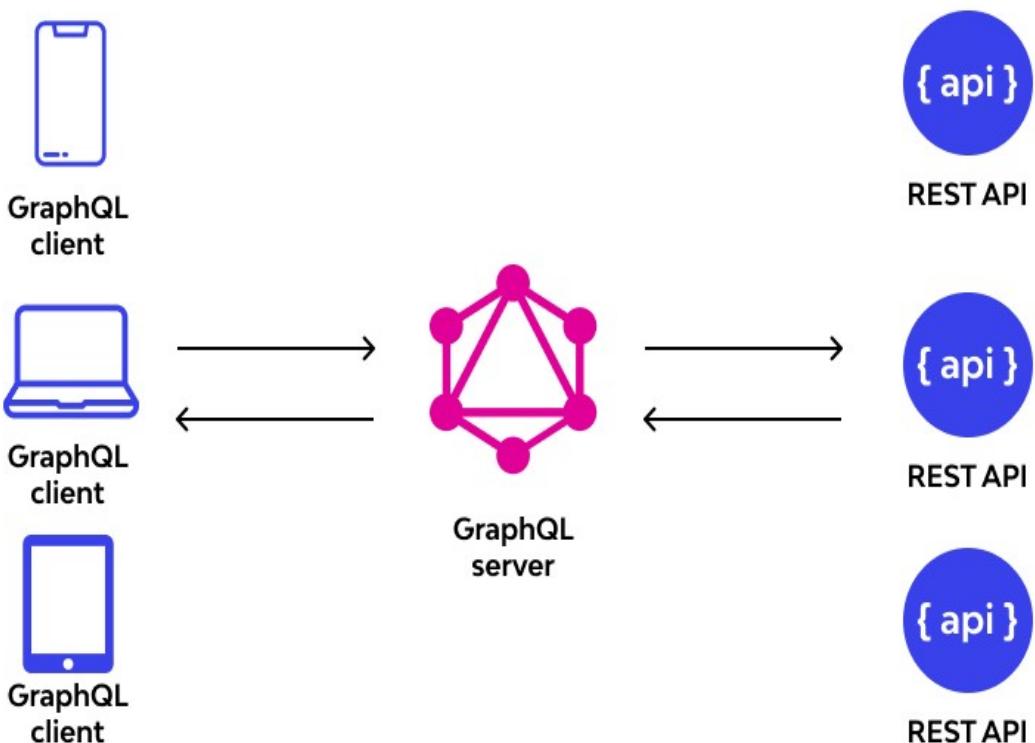
3.3-Documentación del código.

En este apartado veremos como están compuestos los distintos proyectos a nivel de código y la lógica que siguen para poder funcionar correctamente.

3.3.1-Servidor GraphQL.

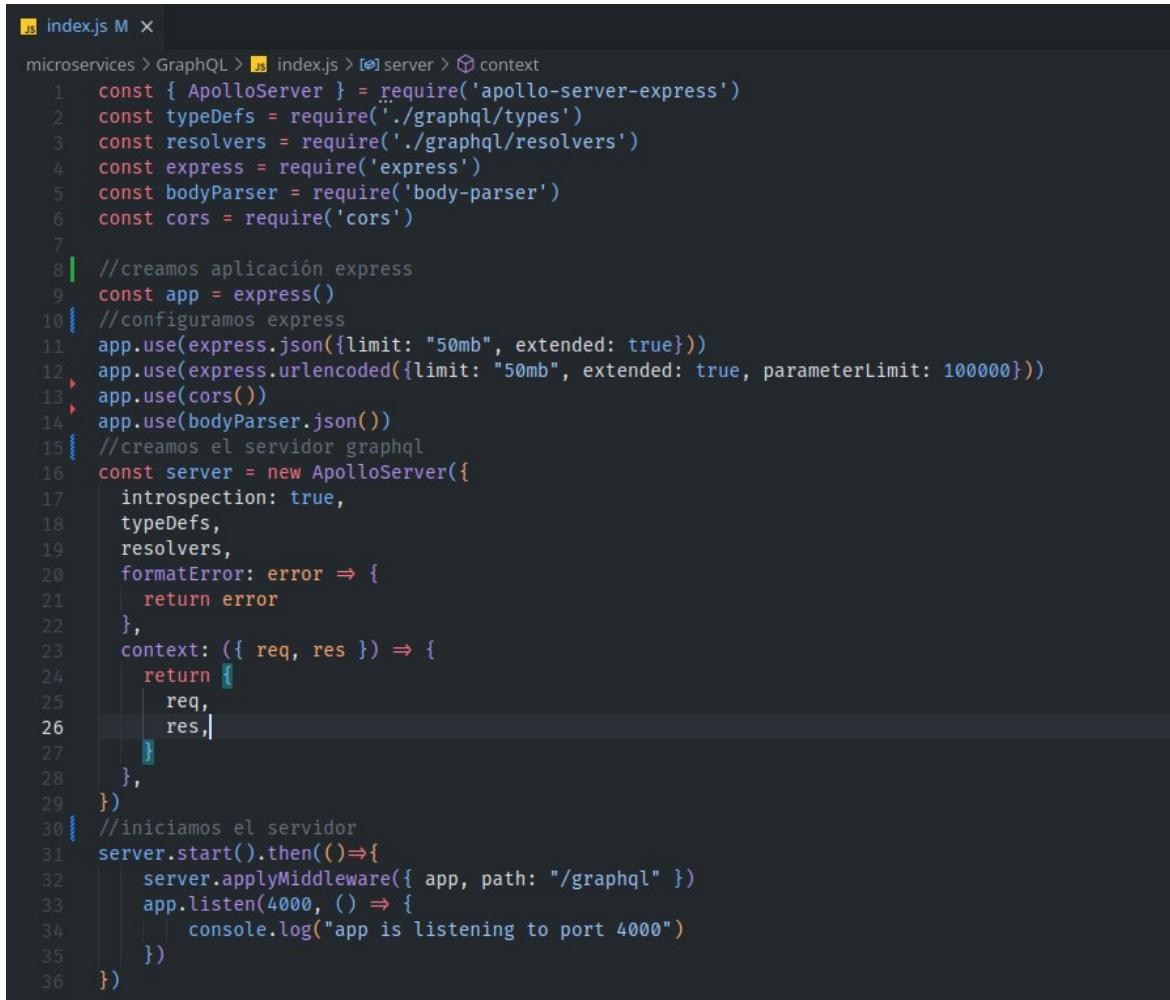
Es el centro de la aplicación, se encarga de gestionar las peticiones del cliente, procesarlas y reenviarlas al microservicio correspondiente. Su principal función es que las peticiones del cliente sean centralizadas y el microservicio que se use sea totalmente transparente.

Para ello usa GraphQL, un lenguaje de consulta (API) basado en peticiones HTTP POST cuya función es brindar al cliente únicamente los datos solicitados por lo que el tráfico de la red, los tiempos de respuesta y la carga de la memoria es menor.



Nuestro servidor GraphQL funciona bajo JavaScript en un servidor Node.js y lo iniciamos usando Express.

Específicamente usamos un servidor GraphQL ApolloServer, ya que en mi parecer es el servidor GraphQL con mejor rendimiento y más fácil de desarrollar para JavaScript.



```

index.js M ×
microservices > GraphQL > index.js > server > context
1 const { ApolloServer } = require('apollo-server-express')
2 const typeDefs = require('../graphql/types')
3 const resolvers = require('../graphql/resolvers')
4 const express = require('express')
5 const bodyParser = require('body-parser')
6 const cors = require('cors')
7
8 // creamos aplicación express
9 const app = express()
10 // configuramos express
11 app.use(express.json({limit: "50mb", extended: true}))
12 app.use(express.urlencoded({limit: "50mb", extended: true, parameterLimit: 100000}))
13 app.use(cors())
14 app.use(bodyParser.json())
15 // creamos el servidor graphql
16 const server = new ApolloServer({
17   introspection: true,
18   typeDefs,
19   resolvers,
20   formatError: error => {
21     return error
22   },
23   context: ({ req, res }) => {
24     return {
25       req,
26       res,
27     }
28   },
29 })
30 // iniciamos el servidor
31 server.start().then(()=>{
32   server.applyMiddleware({ app, path: "/graphql" })
33   app.listen(4000, () => {
34     console.log("app is listening to port 4000")
35   })
36 })

```

Como vemos, el archivo index.js es muy sencillo, solamente iniciamos Express y el servidor Apollo.

Cuando instanciamos nuestro servidor Apollo encontramos que hacemos referencia a dos imports llamados “*typeDefs*” y “*resolvers*”, aquí es donde se cuece todo nuestro servidor. TypeDefs son los tipos de datos que va a usar nuestro servidor, ya que GraphQL usa tipado, mientras que los resolvers son los controladores que le indican al servidor GraphQL como debe funcionar.



```

// creamos el servidor graphql
const server = new ApolloServer({
  introspection: true,
  typeDefs, // Boxed
  resolvers, // Boxed
  formatError: error => {
    return error
  },
  context: ({ req, res }) => {
    return {
      req,
      res,
    }
  },
})

```

3.3.1.1-Fichero typeDefs.

Aquí encontramos los tipos de datos complejos, las queries y las mutaciones (query es una consulta sencilla y mutación es cuando una consulta afecta en la base de datos).

El tipo de dato más sencillo que encontramos es el tipo Error, este tipo se nos devuelve cuando ha ocurrido un error en la aplicación y necesitamos notificar al usuario.

```
type Error {
  error: String!
}
```

Como vemos el tipo error solo nos devuelve un campo error que contiene un String con el error concreto. Podemos encontrar tipos más complejos como por ejemplo el tipo Product:

```
type Product {
  id: ID!
  name: String!
  price: Float!
  category: String!
  description: String!
  features: Features
  image: String!
  images: [String!]!
  brand: String
  type: String
  valorations: [Valoration]
  stars: Float
  sale: Boolean
}
```

Este tipo contiene varios campos, como un campo id obligatorio, un campo nombre, precio, categoría, valoraciones (array de tipo valoration) etc. Los mismos campos que los productos que están almacenados en la base de datos MongoDB.

Estos tipos se usan para que GraphQL pueda saber que información debe enviar en la respuesta de una petición, aunque para las respuestas he creado uniones de tipos para que no devuelva un tipo de dato obligatoriamente.

```
const { gql } = require('apollo-server-express')

const typeDefs = gql`  

  union ProductRes = Error | Product  

  union UserRes = Error | User  

  union CantidadRes = Error | Cantidad  

  union MsgRes = Error | Message  

  union CompraRes = Error | Compra
```

Al principio de la constante tengo una unión llamada ProductRes formada o bien por un producto o bien por un error. Esta se usa cada vez que se quiera enviar un producto como respuesta y de esta forma, poder devolver también un error en el caso que se haya producido uno.

A su vez, estas uniones las usamos como respuestas en queries y mutations como las siguientes:

```
type Query {
  getProducts(
    search: String
    amount: Int
    sale: Boolean
    category: String
    order: String
    price: [Float]
  ): [ProductRes]!

  getProductOne(
    ident: String!
  ): ProductRes
}
```

getProducts → acepta como argumento los distintos filtros de búsqueda para los productos, siempre devuelve un Array de ProductRes, por lo que si la consulta es exitosa devuelve un array de productos, y si es fallida devuelve un array con el error ocurrido.

getProductOne → debes indicarle un id como parámetro y te devolverá un producto con ese id o un error en su defecto.

Las mutaciones siguen la misma lógica, aquí un par de ejemplos:

```
type Mutation {
  createUser(
    name: String!
    lastname: String
    username: String!
    email: String!
    password: String!
    bank_account: String!
  ): UserRes

  addMoney(
    money: Int!
    username: String!
  ): CantidadRes
}
```

createUser → crea un usuario con los parámetros especificados, devuelve un tipo de dato UserRes (Usuario || Error).

addMoney → añade dinero a nuestro monedero, debemos especificarle el usuario al que se le va a añadir el dinero y la cantidad a añadir. Devuelve la cantidad de dinero actual del usuario.

3.3.1.2-Fichero resolvers.

Aquí es donde le damos sentido a los tipos especificados en el fichero typeDefs. Lo que exportamos en este fichero es una constante donde indicamos todos los tipos de datos a devolver por las queries o las mutations (todos los tipos acabados en “Res”) y el código que van a ejecutar las mismas.

```
const resolvers = [
  ProductRes: {
    _resolveType: (object) => {
      if (object.name) {
        return "Product";
      }

      if (object.error) {
        return "Error";
      }

      return null;
    }
  },
]
```

Indicamos que cualquier elemento que tenga un tipo ProductRes sea un Product, si el propio elemento tiene un campo name, o un Error, si el propio elemento tiene un campo error. De esta forma el servidor GraphQL ya sabe cuando catalogar de un tipo o de otro todas las uniones (También debemos especificar el resto de uniones del proyecto).

Para las queries o las mutations simplemente especificamos su nombre y la función que va a ejecutar:

```
Query: {
  getProducts: queries.getProducts,
  getProductOne: queries.getProductOne,
}

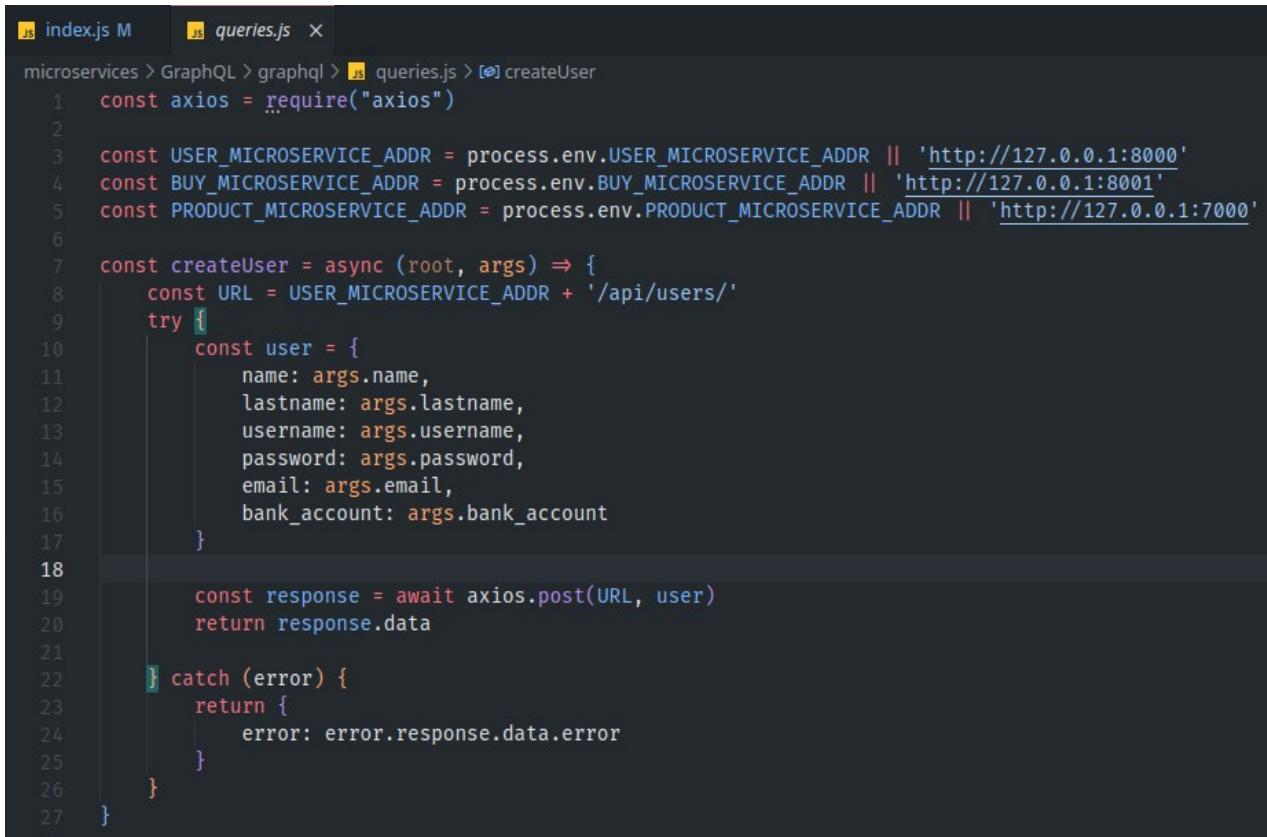
Mutation: {
  createUser: queries.createUser,
  addMoney: queries.addMoney,
  delUser: queries.delUser,
  addValoration: queries.addValoration,
```

Para organizar mi código he separado las funciones de los resolvers en otro fichero llamado queries que veremos a continuación.

3.3.1.3-Funciones del fichero queries.

En el fichero queries encontramos las funciones que ejecutarán las consultas y las mutaciones, estas funciones realizan las peticiones a los microservicios y en algunos casos formatea las respuestas.

Como ejemplo usaremos la función de createUser. Como vemos usamos unas constantes donde guardamos las URLs de los microservicios y en caso de no existir la URL local.



```

index.js M queries.js X
microservices > GraphQL > graphql > queries.js > createUser
1 const axios = require("axios")
2
3 const USER_MICROSERVICE_ADDR = process.env.USER_MICROSERVICE_ADDR || 'http://127.0.0.1:8000'
4 const BUY_MICROSERVICE_ADDR = process.env.BUY_MICROSERVICE_ADDR || 'http://127.0.0.1:8001'
5 const PRODUCT_MICROSERVICE_ADDR = process.env.PRODUCT_MICROSERVICE_ADDR || 'http://127.0.0.1:7000'
6
7 const createUser = async (root, args) => {
8     const URL = USER_MICROSERVICE_ADDR + '/api/users/'
9     try {
10         const user = {
11             name: args.name,
12             lastname: args.lastname,
13             username: args.username,
14             password: args.password,
15             email: args.email,
16             bank_account: args.bank_account
17         }
18
19         const response = await axios.post(URL, user)
20         return response.data
21
22     } catch (error) {
23         return {
24             error: error.response.data.error
25         }
26     }
27 }

```

La función createUser crea un objeto user con los argumentos proporcionados para pasárselos al path “/api/users/” de nuestro microservicio de usuarios mediante una petición POST. Nuestro microservicio se encargará de guardar el usuario en base de datos y en caso exitoso devolverlo o en su defecto retornar el error ocurrido.

Otra función de interés es sendBuy, que se encarga de realizar una compra.

En esta función además de mandar el objeto con la compra en este caso, también enviamos un token en la cabecera de autorización, ya que el microservicio de compras necesita validar que estés logueado y que tu usuario sea correcto, de otra forma podrías realizar comprar para usuarios que no son los tuyos.

```

const sendBuy = async (root, args) => {
  const URL = BUY_MICROSERVICE_ADDR + '/api/compras/'
  try {
    const buy = {
      idUsuario: args.idUsuario,
      precioTotal: args.precioTotal,
      fechaPedido: args.fechaPedido,
      fechaEntrega: args.fechaEntrega,
      articulos: args.articulos
    }
    const config = {
      headers: { Authorization: args.token }
    }
    const response = await axios.post(URL, buy, config)
    return response.data

  } catch (error) {
    return {
      error: "No tienes dinero suficiente para realizar el pedido."
    }
  }
}

```

Sin duda la función más compleja es getProducts ya que necesita formatear los datos antes de mandarlos al microservicio.

```

//función para conseguir los productos
const getProducts = async (root, args) => {
  //cantidad es igual a la pasada por argumento o en su defecto a todos los documentos.
  const amount = args.amount? args.amount: 50
  const maxPrice = args.price? args.price[0]: 1000000
  const minPrice = args.price? args.price[1]: 0
  const searchParam = args.search? args.search: 'no-search'
  const order = args.order? args.order: "Destacados"

  const URL = PRODUCT_MICROSERVICE_ADDR + `/api/products/${amount}/${order}/${maxPrice}/${minPrice}/${searchParam}/${args.sale}`

  if(minPrice > maxPrice){
    return [{error: "Precios incorrectos"}]
  }

  try {
    const response = await axios.get(URL)
    let products = [...response.data]

    return products
  } catch (error) {
    return [
      {
        error: "Productos no encontrados"
      }
    ]
  }
}

```

El microservicio de productos necesita que le llegue una petición estrictamente ordenada para que el buscador y los filtros funcionen correctamente.

Necesitamos enviarle todos los filtros por lo que los filtros que no hayamos especificado los sustituimos por filtros que no apliquen diferencia en nuestra búsqueda, por ejemplo si no indicamos la cantidad de productos, por defecto será 50 que es la cantidad máxima que podemos enviar, si no especificamos el precio máximo o mínimo se sustituirán por un millón y cero, por lo que aparecerán todos los productos.

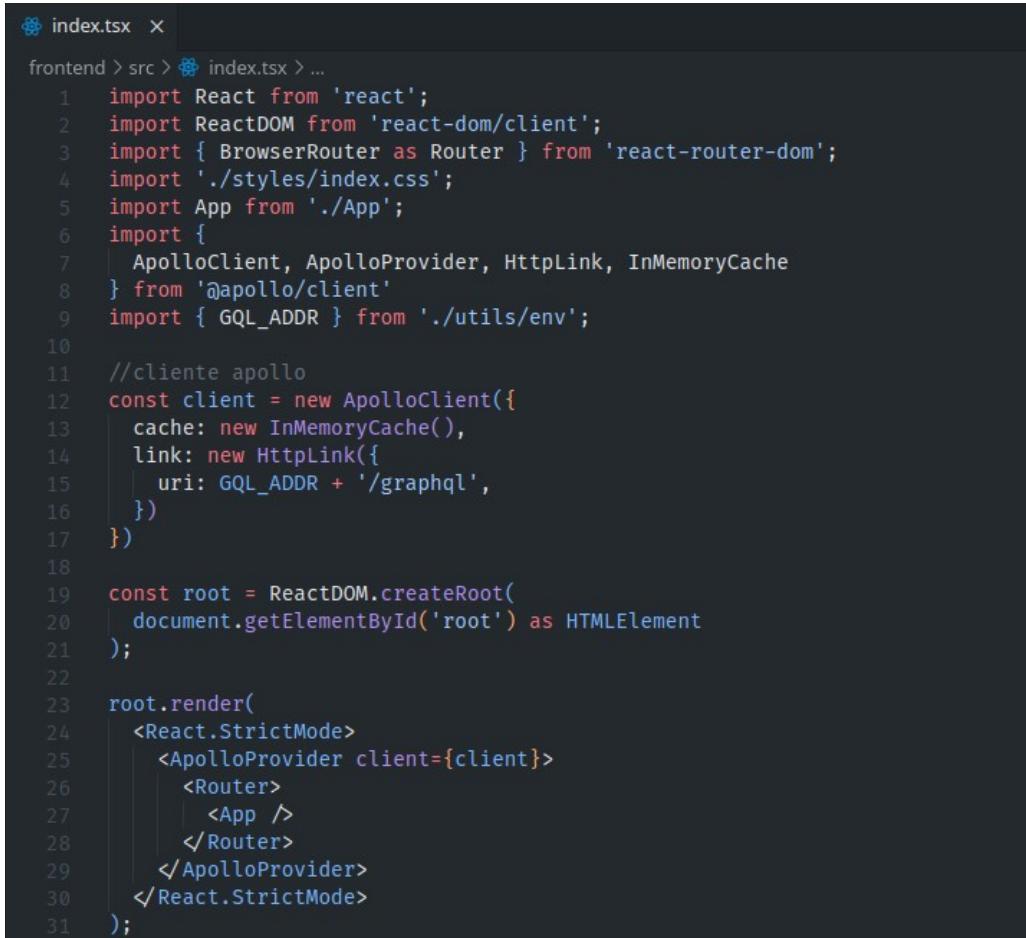
```
const amount = args.amount? args.amount: 50
const maxPrice = args.price? args.price[0]: 1000000
const minPrice = args.price? args.price[1]: 0
const searchParam = args.search? args.search: 'no-search'
const order = args.order? args.order: "Destacados"
```

Todo esto lo especificamos en las ternarias declaradas al principio de la función.

3.3.2-Frontend

El Frontend de nuestra aplicación se encuentra desarrollado con React.js y Typescript, en este punto veremos como está construido y su funcionamiento.

El archivo principal es el index.tsx que tiene la siguiente estructura:



```
index.tsx x
frontend > src > index.tsx > ...
1 import React from 'react';
2 import ReactDOM from 'react-dom/client';
3 import { BrowserRouter as Router } from 'react-router-dom';
4 import './styles/index.css';
5 import App from './App';
6 import {
7   ApolloClient, ApolloProvider, HttpLink, InMemoryCache
8 } from '@apollo/client'
9 import { GQL_ADDR } from './utils/env';
10
11 //cliente apollo
12 const client = new ApolloClient({
13   cache: new InMemoryCache(),
14   link: new HttpLink({
15     uri: GQL_ADDR + '/graphql',
16   })
17 });
18
19 const root = ReactDOM.createRoot(
20   document.getElementById('root') as HTMLElement
21 );
22
23 root.render(
24   <React.StrictMode>
25     <ApolloProvider client={client}>
26       <Router>
27         <App />
28       </Router>
29     </ApolloProvider>
30   </React.StrictMode>
31 );
```

En este fichero estamos renderizando toda nuestra aplicación dentro de un div, gracias a la tecnología de Virtual DOM de React.js, JavaScript se encarga de simular todo el DOM ganando mucha velocidad de ejecución y facilidad de desarrollo.

Dentro de este div añadimos las etiquetas de “Router” y “ApolloProvider” para poder usar el enrutamiento de React.js y GraphQL.

También estamos instanciando el cliente Apollo que nos permitirá hacer consultas a nuestro servidor GraphQL.

Renderizamos el componente “App” que es el que se encarga de gestionar que componente se renderiza dependiendo del Path, mostrar mensajes por pantalla, etc.

3.3.2.1-Componente Home.

El componente home renderiza la pantalla de inicio donde encontramos una vista con algunos de los productos de nuestra tienda, un carrusel con anuncios, etc. La pantalla es la siguiente:



Veamos un poco como esta hecho a nivel de código.

```
const Home: React.FC = () =>{
  const imagenes = [fondo, a, fondo, a]

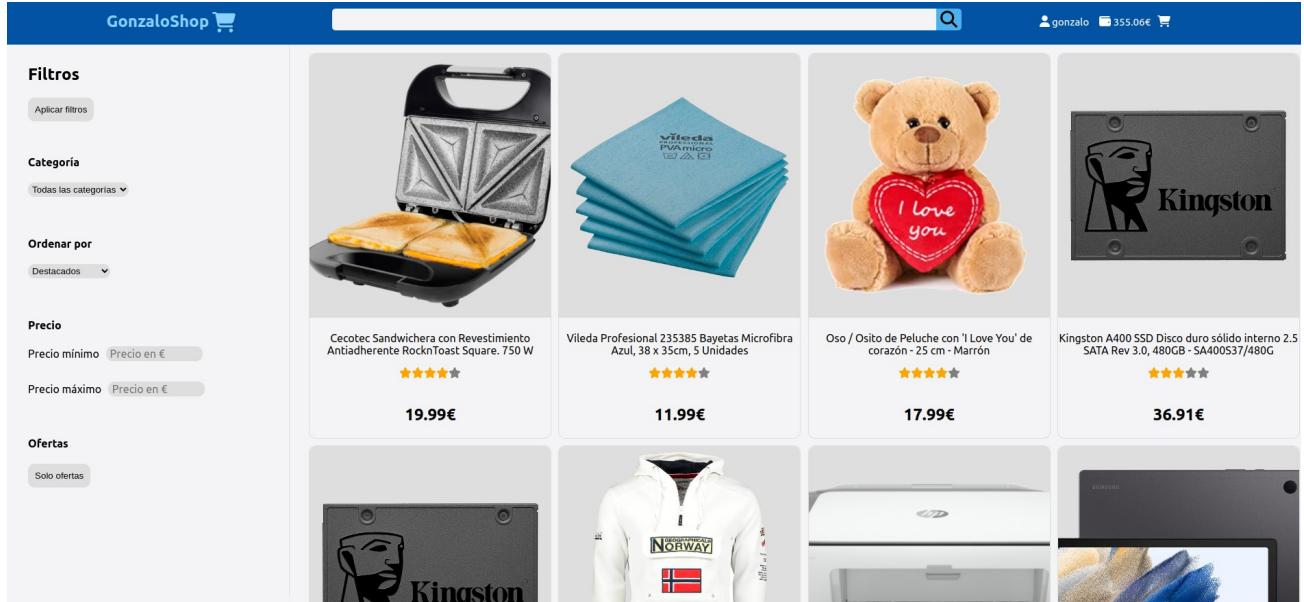
  return(
    <div className="home">
      <section className="center">
        <Slider images={imagenes}/>
      </section>
      <section className="center">
        <Products/>
      </section>
      <section className="center">
        <Link to={'/products'} className="productsLink">
          <h3>Echa un vistazo a todos nuestros productos</h3>
        </Link>
      </section>
      <section className="center">
        <Offerts/>
      </section>

    </div>
  )
}
```

Como vemos el componente principal simplemente son varios componentes ordenados como el componente Slider que contiene el carrusel , el componente Products que contiene la vista de los productos o el componente Offerts que contiene la vista de las ofertas.

3.3.2.2-Componente Products.

El componente Products renderiza una vista con todos los productos donde podemos usar el buscador, aparecen los filtros de búsqueda, etc. La pantalla es la siguiente:



Veamos un poco como esta hecho a nivel de código.

```
return (
  <div className='mb-15'>
    <LateralPanel handleSubmit={handleSubmitFilters} setCategory={setCategory}
      setMaxPrice={setMaxPrice} setMinPrice={setMinPrice}
      setOrder={setOrder} setRadio={setRadio} radio={radio}
      handleSale={handleSale} category={category} order={order}>
    <section className="productsResults">
      {products.map((p:Product, i) => <ProductOne key={i} product={p}/>)}
    </section>
  </div>
);
```

En la estructura encontramos que tiene un componente “*LateralPanel*” que no es más que el panel con los filtros y una sección donde se renderiza cada producto como un componente “*ProductOne*” estos componentes son cada uno de los productos que aparecen por pantalla.

Para conseguir estos productos ejecutamos una consulta GraphQL a nuestro servidor justo cuando renderizamos la pantalla gracias a un UseEffect.

```
const Products:React.FC = () => {

    const [radio, setRadio] = useState<boolean>(false)
    const [category, setCategory] = useState<string>("Todas las categorías")
    const [order, setOrder] = useState<string>("")
    const [maxPrice, setMaxPrice] = useState<number>(100000)
    const [minPrice, setMinPrice] = useState<number>(0)

    const [getProducts, result] = useLazyQuery(PRODUCTS_SEARCH)
    const [products, setProducts] = useState<Product[]>([])
    const {search} = useParams()

    //Effect para hacer la query
    useEffect(() => {
        getProducts({variables: {
            search: search,
            price: [maxPrice, minPrice],
            category: category,
            order: order
        }})
    }, [])
}
```

La consulta ejecutada es la siguiente:

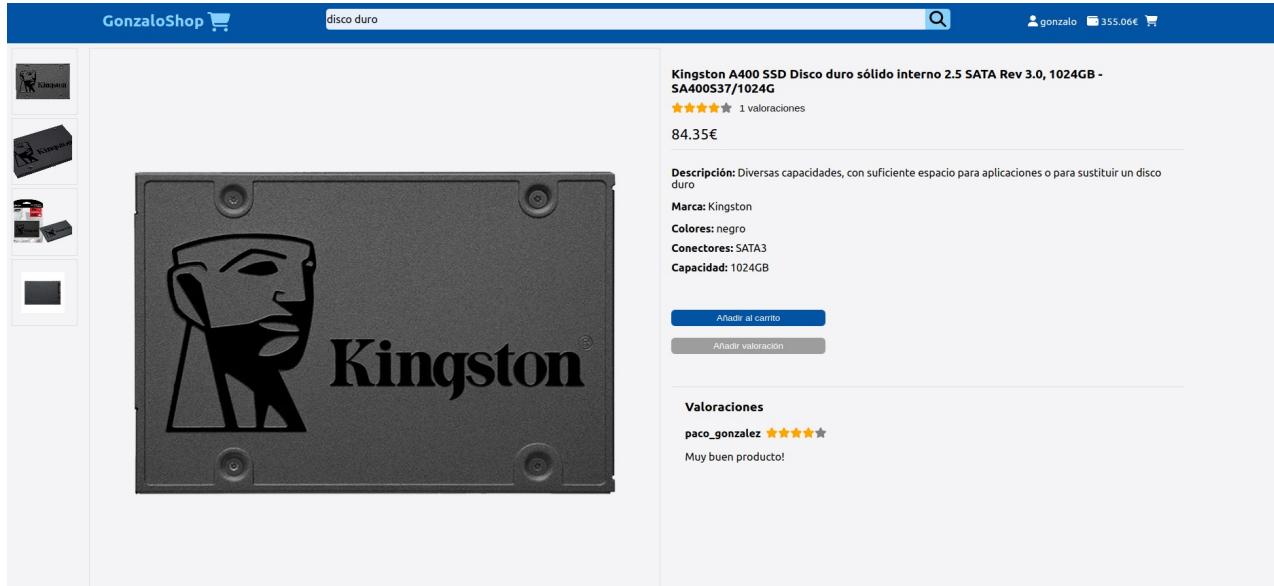
```
export const PRODUCTS_SEARCH = gql`query($search: String, $amount: Int, $sale: Boolean, $price: [Float], $category: String, $order: String){
  getProducts( search: $search amount: $amount sale: $sale price: $price category: $category order: $order){
    __typename
    ... on Product {
      id
      name
      image
      price
      stars
    }
    ... on Error {
      error
    }
  }
}`
```

Dentro de los argumentos le pasamos las variables con los filtros de búsqueda para los productos.

Al aplicar los filtros en el formulario con los filtros de búsqueda simplemente se vuelve a ejecutar esta consulta pero con unos parámetros diferentes.

3.3.2.3-Componente ProductOne.

El componente Products renderiza la vista de un solo producto donde podemos ver distintas imágenes añadirlo al carrito, ver y enviar valoraciones, etc. La pantalla es la siguiente:



Vista del formulario de añadir valoración:

The screenshot shows the 'Añade tu valoración' (Add your review) form. On the left, there's a partial image of the SSD. The form includes fields for writing a review ('Escribe lo que opinas') and selecting a rating ('Selecciona las estrellas'). There are also buttons for 'Añadir al carrito' (Add to cart), 'Cancelar valoración' (Cancel review), and 'Enviar valoración' (Send review).

3.3.2.4-Componente Register.

El componente Register renderiza la vista del formulario para crear un usuario en la aplicación. La pantalla es la siguiente:

The screenshot shows a registration form titled "Crear cuenta" (Create account) from "GonzaloShop". The form contains the following fields:

- Nombre**: Placeholder "Escribe solo el nombre"
- Apellidos**: Placeholder "Escribe aquí tus apellidos"
- Correo electrónico**: Placeholder "Placeholder"
- Número de cuenta**: Placeholder "Placeholder"
- Nombre de usuario**: Placeholder "Nombre de tu usuario de GonzaloShop"
- Contraseña**: Placeholder "Placeholder"
- Repite tu contraseña**: Placeholder "Placeholder"

At the bottom of the form is a blue "Continuar" (Continue) button. Below the button, a note reads: "Todos los campos son obligatorios, pulsa continuar para registrarte." There is also a "Volver" (Back) link.

Un formulario sencillo donde ingresas tus datos para crear una cuenta de GonzaloShop. El código tsx del formulario son solo inputs por lo que voy a omitirlo y centrarme en las validaciones:

Comprobamos que todos los campos obligatorios tengan contenido y que el número de cuenta tenga 20 dígitos.

```
async function handleSubmit(event: FormEvent){
  event.preventDefault()

  try {
    //comprobamos los campos
    if (registerValues.name.length < 0 || registerValues.password.length < 0 || registerValues.lastname.length < 0 || registerValues.email.length < 0 || registerValues.username.length < 0){

      toastError('Todos los campos son obligatorios')

    }else{
      if(registerValues.bank_account.length != 20){
        toastError('El número de cuenta debe tener 20 dígitos')
        return null
      }
    }
  } catch (error) {
    console.error(error)
  }
}
```

Si las valoraciones no muestran un error se ejecutará la consulta GraphQL createUser y si el microservicio de usuarios devuelve algún error se muestra por pantalla a través del catch.

```
    return null
}
createUser({variables: {
  name: registerValues.name,
  username: registerValues.username,
  email: registerValues.email,
  password: registerValues.password,
  bankAccount: registerValues.bank_account,
  lastname: registerValues.lastname
}})
}

//mensaje de error
}catch(error: any){
  toastError('Error al crear el usuario')
}
}
```

Consulta GraphQL ejecutada:

```
export const CREATE_USER = gql`mutation($name: String!, $username: String!, $email: String!, $password: String!, $bankAccount: String!, $lastname: String){
  createUser(name: $name, username: $username, email: $email, password: $password, bank_account: $bankAccount, lastname: $lastname ){
    __typename
    ... on User {
      id
      name
      lastname
      username
      token
      bank_account
      wallet {
        cantidad
        descuento
        limite
      }
    }
    ... on Error {
      error
    }
  }
}
```

3.3.2.5-Componente Cart.

El componente Cart renderiza la vista del carrito de la compra con todos sus productos. La pantalla es la siguiente:

The screenshot shows the 'GonzaloShop' application interface. At the top, there's a logo with the text 'GonzaloShop' and a shopping cart icon. Below it, the main content area is divided into two sections: 'Productos' on the left and 'Carrito de la compra de gonzalo.' on the right.

Productos (Left Side):

- Kingston A400 SSD Disco duro sólido interno 2.5 SATA Rev 3.0, 480GB - SA400S37/480G x2**: An image of a Kingston SSD is shown. Below the image, the price is listed as **Precio: 73.82€**, and there is a blue button labeled **Eliminar producto**.
- Peluche de Kaito x1**: An image of a Kaito plushie is shown. Below the image, the price is listed as **Precio: 15.95€**, and there is a blue button labeled **Eliminar producto**.

Carrito de la compra de gonzalo. (Right Side):

- Productos: 3**
- Precio del carrito: 116.72€**
- Descuento: No**
- Precio final: 116.72€**
- Número de cuenta: 12121*******
- Dirección de envío:** (A text input field with placeholder text 'Introduce tu dirección')
- Seguir comprando** (A light gray button)
- Hacer pedido** (A blue button)

Encontramos una pantalla donde vemos todos los productos que tenemos en nuestro carrito de la compra y la opción de hacer el pedido. En esta pantalla lo más relevante es la llamada a la función de realizar compra:

```
export const ADD_BUY = gql`mutation($token: String!, $idUsuario: Int!, $precioTotal: Float!, $fechaPedido: String!, $fechaEntrega: String!, $descuento: Float!){  sendBuy(token: $token, idUsuario: $idUsuario, precioTotal: $precioTotal, fechaPedido: $fechaPedido, fechaEntrega: $fechaEntrega){    __typename    ... on Compra {      fechaEntrega      fechaEntrega      id      idUsuario      articulos {        nombre        precio        cantidad      }      precioTotal      descuento    }    ... on Error {      error    }  }}`
```

3.3.2.6-Componente Compras.

El último componente es Compras, este renderiza una vista con todos los pedidos realizados e información sobre ellos. La pantalla es la siguiente:

Pedidos de gonzalo:					
Nº	Fecha de la compra	Fecha de entrega	Precio Total	Estado del pedido	Artículos
1	2023-01-20	2023-02-01	3.00€	Entregado	Nombre: producto de prueba Cantidad: 2 Precio: 1€ Nombre: producto de prueba2 Cantidad: 1 Precio: 1€
2	2023-01-20	2023-02-01	3.00€	Entregado	Nombre: producto de prueba Cantidad: 2 Precio: 1€ Nombre: producto de prueba2 Cantidad: 1 Precio: 1€
3	2023-01-20	2023-02-01	3.00€	Entregado	Nombre: producto de prueba Cantidad: 2 Precio: 1€ Nombre: producto de prueba2 Cantidad: 1 Precio: 1€
4	2023-01-20	2023-02-01	3.00€	Entregado	Nombre: producto de prueba Cantidad: 2 Precio: 1€ Nombre: producto de prueba2 Cantidad: 1 Precio: 1€

Este componente simplemente consulta los pedidos y los muestra por pantalla.

3.3.3-Microservicio de Productos.

El microservicio de productos está desarrollado en JavaScript corriendo bajo Node.js, se conecta con una base de datos MongoDB y maneja todos sus endpoints con Express.

3.3.3.1-Controlador de Productos.

Este controlador cuenta con dos endpoints, un GET que nos devuelve un producto específico y un PUT que nos añade una valoración.

El código de los endpoints es el siguiente:

```
//devolver un único producto
productRouter.get('/:id', async (request, response) => {
  const id = request.params.id
  if (!id){
    response.status(500).json({error: "Id no encontrado"})
  }

  try {
    const product = await Product.findById(id)
    response.status(200).json(formatDocumentOne(product))

  } catch (error) {
    response.status(500).json({error: "Producto no encontrado"})
  }
})

//añadir una valoración
productRouter.put('/:id', async (request, response) => {
  const id = request.params.id
  const valoration = request.body

  if (!id || !valoration){
    response.status(500).json({error: "Error al publicar la valoración"})
  }

  try {
    await Product.updateOne({_id: {$eq: id}}, {$push: {valorations: valoration}})
    const document = await Product.findOne({_id: {$eq: id}})
    response.status(200).json(formatDocumentOne(document))

  } catch (error) {
    response.status(500).json({error: "Producto no encontrado"})
  }
})
```

3.3.3.2-Controlador de Buscador.

Este controlador solo cuenta con un endpoint, pero es mucho más complejo que los anteriores. El endpoint se encarga de hacer funcionar el buscador y los filtros de búsqueda.

Antes de instanciar el endpoint he creado varias funciones para filtrar los datos según lo necesitemos:

```
//función para filtrar los productos con descuento
function filtrarSale(data, oferta){
    if (oferta === 'true') return data.filter(product => product.sale)
    return data
}

//función para filtrar por categoría
function filtrarCategoria(data, categoria){
    if (categoria) return data.filter(p => p.category === categoria)
    return data
}

//función para ordenar por precio mas alto
function ordenarPrecioMasAlto(data){
    return data.sort(function(a, b) {
        return b.price - a.price
    })
}

//función para ordenar por precio mas bajo
function ordenarPrecioMasBajo(data){
    return data.sort(function(a, b) {
        return a.price - b.price
    })
}

//función para ordenar por valoración
function ordenarValoracion(data){
    return data.sort(function(a, b) {
        return b.stars - a.stars;
    })
}

//función para aplicar orden
function aplicarOrden(data, orden){
    switch (orden) {
        case ORDEN.PRECIO_MAS_ALTO:
            return ordenarPrecioMasAlto(data)

        case ORDEN.PRECIO_MAS_BAJO:
            return ordenarPrecioMasBajo(data)

        case ORDEN.VALORACION:
            return ordenarValoracion(data)

        default:
            return data
    }
}
```

Haciendo uso de estas funciones podemos aplicar todos los filtros, menos el filtro de buscador, que para el usare la función regex de MongoDB.

Lo primero que hacemos al igual que anteriormente es formatear los parámetros:

```
//devolver varios productos
searchRouter.get('/:amount/:order/:maxPrice/:minPrice/:search/:sale/:category', async (request, response) => {
  const { search, amount, order, maxPrice, minPrice, sale, category } = request.params

  //mediante una ternaria igualamos a false el parametro search en el caso que no se este buscando
  const searchQuery = search !== "no-search"
    ? search
    : false

  //mediante una ternaria igualamos a false el parametro search en el caso que no se este buscando
  const categoryQuery = category !== "Todas las categorías"
    ? category
    : false

  //mediante una ternaria comprobamos que la cantidad sea mayor de 40, esto significa que no estamos filtrando
  //es true el valor del campo amount es el de todos los documentos de la base de datos, mientras que si es falso
  const amountQuery = parseInt(amount) >= 40
    ? await Product.collection.countDocuments()
    : parseInt(amount)

  //recogemos el valor del precio máximo y mínimo
  const maxPriceQuery = parseFloat(maxPrice)
  const minPriceQuery = parseFloat(minPrice)
```

Una vez tengamos los parámetros formateados realizamos una consulta si estamos aplicando búsqueda y otra si no lo estamos haciendo.

```
//si estamos usando el buscador ejecutamos la siguiente consulta
if (searchQuery){
  const products = await Product.find(
    {
      $and:[
        {
          $or:[
            {"name": { $regex: searchQuery, $options: 'i' }},
            {"category": { $regex: searchQuery, $options: 'i' }},
            {"type": { $regex: searchQuery, $options: 'i' }}
          ]
        },
        {"price":{$gte: minPriceQuery}},
        {"price":{$lte: maxPriceQuery}}
      ]
    }
  ).limit(amountQuery)
  //aplicamos todos los filtros y ordenamos
  response.status(200).json(
    formatDocuments(
      aplicarOrden(
        filtrarCategoria(
          filtrarSale(products, sale), categoryQuery
        ), order
      )
    )
  )
}
```

Una vez que tenemos los documentos de mongo aplicamos los filtros y devolvemos el array.

```
//si no estamos usando el buscador ejecutamos la siguiente consulta
} else{
    const products = await Product.find({
        $and:[
            {
                "price":{$gte: minPriceQuery}
            },{
                "price":{$lte: maxPriceQuery}
            }
        ]
    }).limit(amountQuery)

    //aplicamos todos los filtros y ordenamos
    response.status(200).json(
        formatDocuments(
            aplicarOrden(
                filtrarCategoria(
                    filtrarSale(products, sale), categoryQuery
                ), order
            )
        )
    )
}

//ejecutamos catch en caso de error
} catch (error) {
    response.status(500).json({error: "Error al encontrar los productos deseados"})
}

module.exports = searchRouter
```

3.3.3.3-Campo estrellas.

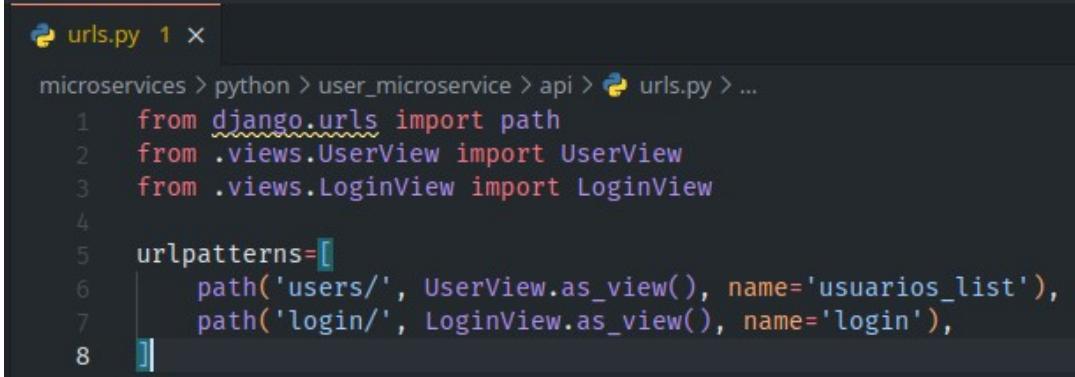
Un punto a tomar en cuenta es el campo estrellas del producto, que no puede ser un campo guardado intrínsecamente en la base de datos ya que es un campo que se calcula haciendo una media de todas las valoraciones.

Para solucionar este problema he hecho uso de los campos virtuales de mongoose. Los campos virtuales no son más que campos calculados que no se encuentran dentro de la base de datos pero actúan como campos normales dentro de mongoose.

```
//campo calculado stars
schema.virtual('stars').get(function(){
    const valorations = { ...this.valorations}
    const starsTotal = valorations.stars.reduce((a, c) => a + c, 0)
    const stars = (starsTotal/valorations.stars.length)
    if(isNaN(stars)){
        return 0
    }else{
        return stars
    }
})
```

3.3.4-Microservicio de Usuarios.

El microservicio de usuarios está desarrollado en Python con el framework de Django y se conecta con una base de datos MySQL. Este microservicio cuenta con dos controladores:



```

urls.py 1 x

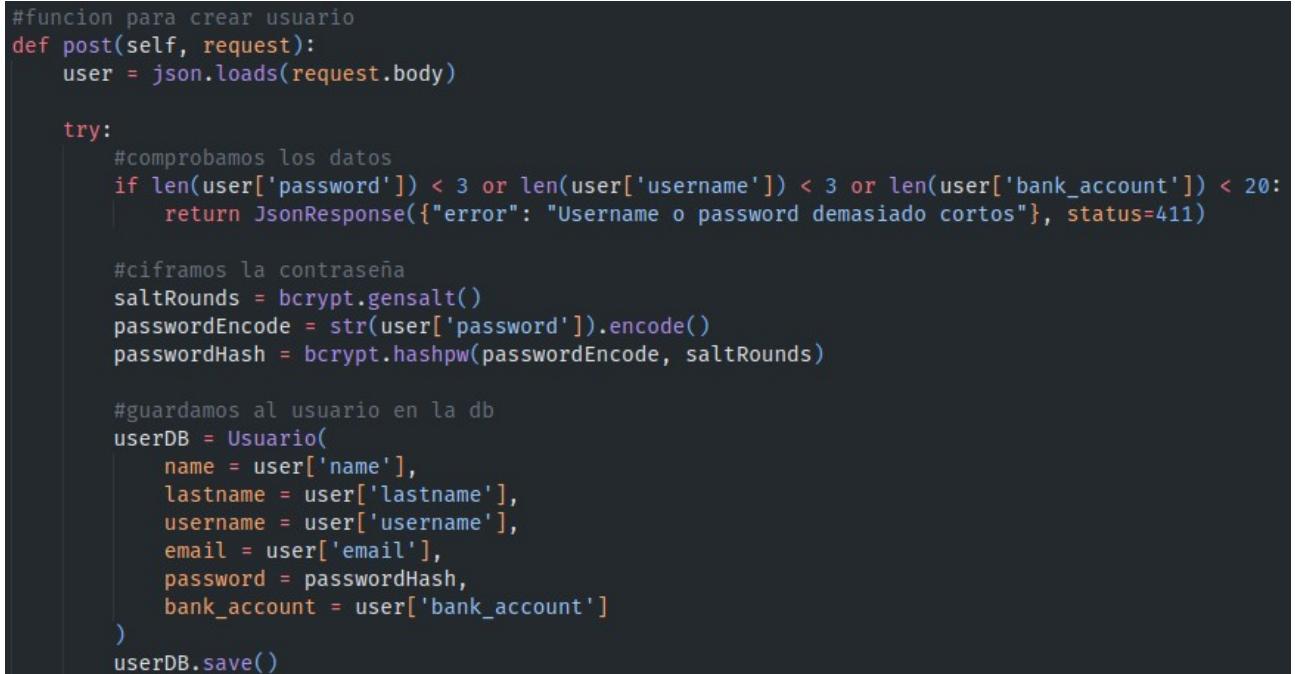
microservices > python > user_microservice > api > urls.py > ...
  1  from django.urls import path
  2  from .views.UserView import UserView
  3  from .views.LoginView import LoginView
  4
  5  urlpatterns=[
  6      path('users/', UserView.as_view(), name='usuarios_list'),
  7      path('login/', LoginView.as_view(), name='login'),
  8

```

El primer controlador es para la creación de usuarios y el segundo para el login de usuarios.

3.3.4.1-Controlador de users.

Este controlador nos permite crear y eliminar usuarios además de añadir dinero a su monedero. Veamos la función de crear usuarios:



```

#funcion para crear usuario
def post(self, request):
    user = json.loads(request.body)

    try:
        #comprobamos los datos
        if len(user['password']) < 3 or len(user['username']) < 3 or len(user['bank_account']) < 20:
            return JsonResponse({"error": "Username o password demasiado cortos"}, status=411)

        #ciframos la contraseña
        saltRounds = bcrypt.gensalt()
        passwordEncode = str(user['password']).encode()
        passwordHash = bcrypt.hashpw(passwordEncode, saltRounds)

        #guardamos al usuario en la db
        userDB = Usuario(
            name = user['name'],
            lastname = user['lastname'],
            username = user['username'],
            email = user['email'],
            password = passwordHash,
            bank_account = user['bank_account']
        )
        userDB.save()
    
```

El primer paso dentro de esta función son comprobar los valores que nos llegan desde el cliente y si hay alguno erróneo, devolver un error y cortar la ejecución de la función.

Si los campos son correctos pasamos a cifrar la contraseña, crear un nuevo objeto de tipo usuario y guardarlo en la base de datos.

```
#Creamos su monedero
wallet = Monedero(
    usuario = userDB,
    cantidad = 0,
    limite = 1000,
    descuento = 20
)
wallet.save()

#Creamos el token
userForToken = {
    'username': userDB.username,
    'id': userDB.id
}

token = jwt.encode(userForToken, os.environ['TOKEN'], algorithm="HS256")

responseData = {
    'id': userDB.id,
    'name': userDB.name,
    'lastname': userDB.lastname,
    'username': userDB.username,
    'bank_account': userDB.bank_account,
    'wallet':{
        'cantidad': wallet.cantidad,
        'limite': wallet.limite,
        'descuento': wallet.descuento
    },
    'token': token
}

return JsonResponse(responseData, status=201)
```

El siguiente paso es crear el monedero del usuario y un token para que pueda identificare, montamos el objeto para responder al cliente y se lo mandamos con un estado 201.

Mediante el uso de excepciones comprobamos los campos únicos de la base de datos como el correo electrónico o el nombre de usuario.

```
#except para usernames y emails ya guardados en la base de datos.
except IntegrityError as e:
    error = str(e.args)
    print(error)

    if "email" in error:
        return JsonResponse({"error": "El email está en uso"}, status=500) Hacer nueva

    elif "username" in error:
        return JsonResponse({"error": "El nombre de usuario está en uso"}, status=500)

    else:
        return JsonResponse({"error": "Error en los datos"}, status=500)
```

La siguiente función que veremos será la eliminación de un usuario:

```
#función para eliminar un usuario
def delete(self, request):
    try:
        token = request.headers['Authorization']
        decodedToken = jwt.decode(token, os.environ['TOKEN'], algorithms=["HS256"])

        if not token or not decodedToken['id']:
            return JsonResponse({"error": "Token inválido o ausente"}, status=401)

        user = Usuario.objects.filter(id = decodedToken['id']).first()

        if not user:
            return JsonResponse({"error": "Token inválido"}, status=401)

        monedero = Monedero.objects.filter(usuario_id = decodedToken['id']).first()

        if not monedero:
            return JsonResponse({"error": "Problemas con el monedero"}, status=401)

        monedero.delete()
        user.delete()

        return JsonResponse({"msg": "Eliminado correctamente"}, status=200)

    except Exception as e:
        print(e)
        return JsonResponse({"error": "Error al eliminar el usuario"}, status=500)
```

Lo primero que hacemos es la comprobación del token y que el usuario que ha generado ese token realmente exista en la base de datos, seguidamente recogemos su monedero para eliminarlo junto con el usuario.

Por último veremos la función de añadir dinero al monedero:

```
#función para añadir dinero al monedero
def put(self, request):
    req = json.loads(request.body)
    try:
        user = Usuario.objects.filter(username = req['username']).first()
        userWallet = Monedero.objects.filter(usuario_id = user.pk).first()

        #comprobamos límite del monedero
        if userWallet.cantidad + req['money'] > 1000:
            return JsonResponse({"error": "Límite de monedero alcanzado"}, status=500)

        userWallet.cantidad = userWallet.cantidad + req['money']
        userWallet.save()

        responseData = {
            'cantidad': userWallet.cantidad
        }

        return JsonResponse(responseData, status=200)

    except Exception as e:
        print(e)
        return JsonResponse({"error": "Error al realizar la transferencia"}, status=500)
```

Para ello traemos el usuario y el monedero de la base de datos, comprobamos que el monedero no haya llegado a su límite y si no es así le sumamos la cantidad enviada por el cliente.

3.3.4.2-Controlador de login.

Este controlador únicamente tiene una función, y es la de loguear a un usuario en nuestra aplicación.

```
#funcion para loguear usuario
def post(self, request):
    userRequest = json.loads(request.body)

    try:
        userDB = Usuario.objects.filter(username = userRequest['username']).first()

        if not userDB:
            return JsonResponse({'error': 'Usuario no existente'}, status=500)

        password = str(userRequest['password']).encode()
        hashed = userDB.password
```

Lo primero que hacemos en esta función es comprobar si en nuestra base de datos existe un usuario con el nombre de usuario especificado y si no es así, enviamos un error al cliente.

```
if bcrypt.checkpw(password, hashed):
    #Creamos el token
    userForToken = {
        'username': userDB.username,
        'id': userDB.id
    }
    token = jwt.encode(userForToken, os.environ['TOKEN'], algorithm="HS256")

    userWallet = Monedero.objects.filter(usuario_id = userDB.pk).first()

    responseData = [
        'id': userDB.id,
        'name': userDB.name,
        'lastname': userDB.lastname,
        'username': userDB.username,
        'bank_account': userDB.bank_account,
        'wallet':{
            'cantidad': userWallet.cantidad,
            'limite': userWallet.limite,
            'descuento': userWallet.descuento
        },
        'token': token
    ]

    return JsonResponse(responseData, status=201)
else:
    return JsonResponse({'error': 'Las contraseñas no coinciden'}, status=406)
```

Lo siguiente es comprobar el hash de la contraseña del usuario en base de datos con la contraseña que ha enviado el cliente, si el procedimiento es exitoso generamos un token y enviamos la respuesta al cliente, si no lo es enviamos el error correspondiente.

3.3.5-Microservicio de Compras.

El microservicio de compras está desarrollado en PHP con el framework de Laravel y se conecta con una base de datos MySQL. Este microservicio cuenta con un único controlador que gestiona las compras.

3.3.5.1-Controlador de compras.

Este controlador cuenta con dos funciones, una para realizar una compra y otra para ver las compras de un usuario. Veamos la función de crear compra:

```
//función para post
public function store(Request $request){
    if(!$request→header('Authorization')){
        return abort(401, 'Debe proveer un Token ');
    }

    $tokenID = $request→idUsuario;
    $sql = "SELECT au.id, am.cantidad, am.descuento FROM api_usuario au
        LEFT JOIN api_monedero am
        ON au.id = am.usuario_id
        WHERE au.id = $tokenID;";

    $user = DB::select($sql);

    if(count($user) != 1){
        return abort(401, 'El usuario no existe');
    }
}
```

En la primera parte de la función comprobamos el token y que el usuario exista en la base de datos.

```
$applyDiscount = ($user[0]→descuento != 0);
$precio = $applyDiscount
    ?($request→precioTotal * $user[0]→descuento)/100
    :$request→precioTotal;
$dineroTotal = $user[0]→cantidad - $precio;

if($dineroTotal < 0){
    return abort(500, 'Dinero insuficiente para realizar la compra');
}
```

Comprobamos si el usuario tiene descuento y si es el caso se lo aplicamos, comprobamos el dinero que le quedaría al usuario después de realizar la compra, y si es menos de cero devuelve un error.

```

$newCompra = new Compras;
$newCompra->idUsuario = $tokenID;
$newCompra->precioTotal = $request->precioTotal;
$newCompra->fechaPedido = $request->fechaPedido;
$newCompra->fechaEntrega = $request->fechaEntrega;
$newCompra->save();

foreach ($request->articulos as $articulo) {
    $newArticulo = new Articulo;
    $newArticulo->precio = $articulo['precio'];
    $newArticulo->nombre = $articulo['nombre'];
    $newArticulo->cantidad = $articulo['cantidad'];
    $newArticulo->compra = $newCompra->id;
    $newArticulo->save();
}

$payment = "UPDATE api_monedero SET cantidad=$dineroTotal, descuento=0 WHERE usuario_id = $tokenID;";
DB :: update($payment);

$articulos = DB :: table('articulos')->where('compra', '=', $newCompra->id)->get();
$compra = array(
    "id"=>$newCompra->id,
    "idUsuario"=>$newCompra->idUsuario,
    "precioTotal"=>$newCompra->precioTotal,
    "fechaPedido"=>$newCompra->fechaPedido,
    "fechaEntrega"=>$newCompra->fechaEntrega,
    "articulos"=>$articulos,
    "descuento"=> $applyDiscount
);
return $compra;

```

Si todo ha ido correctamente guardamos la compra y todos sus artículos, efectuamos el pago y devolvemos la compra.

Veamos por último la función de ver todas las compras de un usuario:

```

//función para get con parámetros
public function show($id, Request $request){
    if(!$request->header('Authorization')){
        return abort(401, 'Debe proveer un Token ');
    }

    $sql = "SELECT id, name, username FROM api_usuario
            WHERE id = $id;";

    $user = DB :: select($sql);

    if(count($user) != 1){
        return abort(401, 'El usuario no existe');
    }

    $compras = DB :: table('compras')->where('idUsuario', '=', $id)->get();
    $response = array();

```

Comprobamos el token y el usuario, si todo está en orden recogemos las compras de la base de datos, las guardamos en un array y las enviamos al cliente.

```
foreach ($compras as $cmp) {  
    $articulos = DB::table('articulos')->where('compra', '=', $cmp->id)->get();  
    $compra = array(  
        "id"=>$cmp->id,  
        "idUsuario"=>$cmp->idUsuario,  
        "precioTotal"=>$cmp->precioTotal,  
        "fechaPedido"=>$cmp->fechaPedido,  
        "fechaEntrega"=>$cmp->fechaEntrega,  
        "articulos"=>$articulos  
    );  
    array_push($response, $compra);  
}  
  
return $response;
```

3.4-Documentación del desarrollo del despliegue.

Como comentamos anteriormente la aplicación será desplegada con Kubernetes, para ello necesitamos una imagen para ejecutar en nuestros pods. Yo he decidido crear las imágenes con Docker además de un entorno de pruebas de las mismas usando un fichero docker-compose.yml, de esta forma podré desarrollar las imágenes de cada contenedor y probar que funcionen correctamente antes de subirlas a Docker Hub.

3.4.1-Creación de las imágenes con Docker.

He creado una imagen autodesplegable por cada componente de la aplicación (Frontend, GraphQL, Microservicios y Bases de datos) esto significa que solo con lanzar los contenedores tendremos la aplicación funcionando automáticamente sin necesidad de hacer ningún cambio más.

Para la creación de las imágenes necesito de un fichero Dockerfile y en algunos casos un script entrypoint.sh para ejecutar algunas acciones cuando inicie el contenedor.

3.4.1.1-Creación de la imagen del Frontend.

Para el despliegue del Frontend cree el siguiente Dockerfile:

```
1  FROM node:19.6.0-bullseye
2
3  # Actualizamos e instalamos lo necesario para correr nuestra aplicacion
4  RUN apt update
5  RUN apt install nano git curl apache2 -y
6
7  # Clonamos el repositorio de GitHub
8  RUN git clone https://github.com/GonzaloRando03/GonzaloShop_Frontend.git
9
10 # Añadimos constante con la dirección del servidor GQL en el fichero env
11 RUN echo "export const GQL_ADDR = \"http://127.0.0.1:4500\"> GonzaloShop_Frontend/src/utils/env.ts
12
13 # Copiamos el entrypoint
14 COPY ./entrypoint.sh /
```

Usé como base una imagen de node corriendo en Debian. Sobre esta imagen instalamos los paquetes necesarios para correr la aplicación, uno de ellos un servidor apache que será donde despleguemos nuestro Frontend, nos bajamos el código de nuestro repositorio de GitHub, añadimos una variable de entorno y copiamos el fichero entrypoint.sh que contiene una serie de órdenes para ejecutar en cada inicio del contenedor.

El entrypoint.sh contiene lo siguiente:

```

1  #!/bin/sh
2  # Añadimos constante con la dirección del servidor GQL en el fichero env
3  if [ $1 ]; then
4      echo "export const GQL_ADDR = \"$1\"">> GonzaloShop_Frontend/src/utils/env.ts
5  fi
6
7  service apache2 start
8  cd GonzaloShop_Frontend/
9  npm install
10 npm run build
11 cp -r build/* /var/www/html/
12 service apache2 reload
13 tail -f /dev/null

```

Comprobamos si pasamos un parámetro, si es así lo cambiamos por la constante de la dirección del servidor GraphQL. Seguidamente iniciamos el servicio apache, compilamos el código del Frontend, lo movemos a la carpeta para su despliegue y reiniciamos el servicio de apache para que muestre nuestro programa.

Con estos dos ficheros tendríamos desplegado nuestro Frontend.

3.4.1.2-Creación de la imagen del servidor GraphQL.

Para el despliegue del servidor GraphQL cree el siguiente Dockerfile:

```

1  FROM node:19.6.0-bullseye
2  RUN apt update
3  RUN apt install nano git curl -y
4  RUN git clone https://github.com/GonzaloRando03/GonzaloShop_ServerGraphQL.git
5  COPY ./entrypoint.sh /

```

Como base usamos la misma imagen que en el Frontend, instalamos los paquetes necesarios, descargamos nuestro código de GitHub y copiamos el archivo entrypoint.sh.

En este script instalamos nuestro proyecto y lo arrancamos.

```

1  #!/bin/bash
2  cd GonzaloShop_ServerGraphQL/
3  npm install
4  node index.js
5  tail -f /dev/null

```

3.4.1.3-Creación de la imagen del microservicio de Productos.

Para el despliegue del microservicio de productos cree el siguiente Dockerfile:

```

1  FROM node:19.6.0-bullseye
2
3  # Actualizamos e instalamos todo lo necesario
4  RUN apt update
5  RUN apt install nano git curl -y
6
7  # Clonamos el repositorio de GitHub
8  RUN git clone https://github.com/GonzaloRando03/GonzaloShop_Product_Microservice.git
9
10 # Copiamos el entrypoint
11 COPY ./entrypoint.sh /

```

Como base usamos la misma imagen que en el Frontend y en el servidor GraphQL, instalamos los paquetes necesarios para iniciar el microservicio, descargamos nuestro código de GitHub y copiamos el archivo entrypoint.sh.

En este script instalamos nuestro proyecto y lo arrancamos.

```

1 #!/bin/bash
2 cd GonzaloShop_Product_Microservice/
3 npm install
4 node index.js
5 tail -f /dev/null

```

3.4.1.4-Creación de la imagen del microservicio de Usuarios.

Para el despliegue del microservicio de usuarios cree el siguiente Dockerfile:

```

1  FROM python:3.12-rc-bullseye
2
3  # Actualizamos e instalamos los paquetes necesarios
4  RUN apt-get update
5  RUN apt-get install nano git curl cargo default-libmysqlclient-dev build-essential -y
6
7  # Clonamos el código del repositorio de GitHub
8  RUN git clone https://github.com/GonzaloRando03/GonzaloShop_User_Microservice.git /deploy/app
9
10 WORKDIR /deploy/app
11
12 # Instalamos las dependencias
13 RUN pip install -r requirements
14
15 # Copiamos el entrypoint
16 COPY ./entrypoint.sh /

```

Usé como base una imagen de python corriendo en Debian. Sobre esta imagen instalamos los paquetes necesarios para correr la aplicación, nos bajamos el código de nuestro repositorio de GitHub, instalamos las dependencias de nuestro proyecto y copiamos el fichero entrypoint.sh.

El fichero entrypoint.sh realiza las siguientes acciones:

```

1 #!/bin/bash
2 # Ejecutamos sleep para que la base de datos mysql tenga tiempo para arrancar y realizar la migración
3 sleep 60
4
5 cd /deploy/app
6
7 # Hacemos las migraciones
8 python3 manage.py makemigrations
9 python3 manage.py migrate
10
11 # Iniciamos el servidor
12 python3 manage.py runserver 0.0.0.0:8000
13 tail -f /dev/null

```

Primero realizamos un sleep de un minuto para que el contenedor de MySQL tenga tiempo para arrancarse ya que si ejecutamos las migraciones y este no está levantado nuestro microservicio no podría conectarse a la base de datos y tendríamos que hacerlo manualmente. Después del sleep ejecutamos las migraciones y arrancamos el servidor.

3.4.1.5-Creación de la imagen del microservicio de Compras.

El despliegue de una aplicación Laravel con Docker es un tanto complicado, por lo que el fichero Dockerfile es algo extenso. Para el despliegue del microservicio de compras cree el siguiente fichero:

```

1 FROM php:8.1.9-fpm-alpine
2
3 # Actualizamos e instalamos los paquetes necesarios
4 RUN apk --no-cache upgrade && \
5     apk --no-cache add bash git sudo openssh libssh2 libxml2-dev oniguruma-dev autoconf gcc g++ make npm freetype-dev
6
7 # Instalamos las extensiones de PHP
8 RUN pecl channel-update pecl.php.net
9 RUN pecl install pcov swoole
10 RUN docker-php-ext-configure gd --with-freetype --with-jpeg
11 RUN docker-php-ext-install mbstring xml pcntl gd zip sockets pdo pdo_mysql bcmath soap
12 RUN docker-php-ext-enable mbstring xml gd zip pcov pcntl sockets bcmath pdo pdo_mysql soap swoole
13 RUN docker-php-ext-install pdo pdo_mysql sockets
14 RUN curl -sS https://getcomposer.org/installer | php -- \
15     --install-dir=/usr/local/bin --filename=composer
16

```

Usamos como base una imagen de PHP corriendo sobre alpine. Como primer paso instalamos todos los paquetes necesarios y las extensiones de PHP que necesita nuestro microservicio.

```
16  
17 # Añadimos composer a nuestro contenedor  
18 COPY --from=composer:latest /usr/bin/composer /usr/bin/composer  
19 COPY --from=spiralscout/roadrunner:2.4.2 /usr/bin/rr /usr/bin/rr  
20  
21 # Clonamos el proyecto de GitHub  
22 RUN git clone https://github.com/GonzaloRando03/GonzaloShop_Buy_Microservice.git .  
23  
24 # Instalamos el proyecto  
25 RUN composer install  
26 RUN composer require laravel/octane spiral/roadrunner  
27 RUN npm install --global yarn  
28 RUN yarn  
29 COPY ./env .  
30  
31 # Realizamos las migraciones y preparamos el despliegue  
32 RUN php artisan key:generate  
33 RUN php artisan octane:install --server="swoole"  
34  
35 # Copiamos el entrypoint  
36 COPY ./entrypoint.sh /  
37 COPY ./env /GonzaloShop_Buy_Microservice/  
38  
39  
40 EXPOSE 8000
```

Después debemos añadir composer a nuestra imagen, descargar nuestro código de GitHub e instalar nuestro proyecto, configuramos el despliegue y copiamos los archivos necesarios.

En el entrypoint.sh ejecutamos las siguientes ordenes:

```
1 #!/bin/bash  
2 # Ejecutamos sleep para que la base de datos mysql tenga tiempo para arrancar y realizar la migración  
3 sleep 60  
4 # Realizamos las migraciones e iniciamos el servidor  
5 cd /var/www/html/  
6 php artisan migrate  
7 php artisan octane:start --server="swoole" --host="0.0.0.0"  
8 tail -f /dev/null
```

Al igual que con el microservicio de usuarios debemos ejecutar el sleep, seguidamente realizamos las migraciones y arrancamos nuestro microservicio.

3.4.1.6-Creación de la imagen de la base de datos MySQL.

Para el despliegue de la base de datos MySQL simplemente usé la imagen oficial de MySQL:

```
1     FROM mysql:8.0.13
```

3.4.1.7-Creación de la imagen de la base de datos MongoDB.

Para el despliegue de la base de datos MongoDB simplemente cree el siguiente Dockerfile:

```
1     FROM mongo:4.2.9-bionic
2
3     #Copiamos el script de para preparar nuestra base de datos
4     COPY ./initGonzaloShopDB.sh /
5     RUN ./initGonzaloShopDB.sh 1> run.log 2> error.log
```

Usé como base la imagen oficial de MongoDB a la que le aplico un script para crear los usuarios y bases de datos. El script contiene las siguientes ordenes:

```
1 #!/bin/bash
2  # Eliminamos los datos de la base de datos en caso de que exista
3  echo -e "use $MONGO_INITDB_DATABASE \n db.products.drop();" "\n""exit" | mongo
4
5  # Añadimos datos a la base de datos
6  echo Añadiendo datos a la base de datos
7  mongoimport --db $MONGO_INITDB_DATABASE --collection products --file /etc/mongo/data.json --jsonArray
8  echo Datos añadidos
9
10 # Creamos los usuarios con los que nos conectaremos a la base de datos
11 echo -e "use admin \n db.createUser({user: \"root\", pwd: \"$MONGO_INITDB_PASSWORD\","
12     roles: [ { role: \"userAdminAnyDatabase\", db: \"admin\" } ]});" "\n""exit" | mongo
13
14 echo -e "use $MONGO_INITDB_DATABASE \n db.createUser({ user: \"$MONGO_INITDB_USERNAME\", pwd: \"$MONGO_INITDB_PASSWORD\",
15     roles: [ { \"role\": \"readWrite\", \"db\": \"$MONGO_INITDB_DATABASE\" } ] });" "\n""exit" | mongo
16
17 exit 0
```

Eliminamos la colección products de nuestra base de datos en caso de que exista, añadimos los datos a nuestra base de datos y creamos los usuarios para poder administrar y conectarnos a la base de datos.

3.4.1.8-Imágenes en Docker Hub.

Para poder usar las imágenes desde cualquier lugar sin necesidad de tener todos los Dockerfiles subí las imágenes a Docker Hub una vez comprobé que la aplicación funcionaba correctamente.

The screenshot shows the Docker Hub interface with the search bar set to 'gonzalors23'. Six repositories are listed:

- gonzalors23 / gonzaloshop-user-microservice**
Contains: Image | Last pushed: 12 days ago
Inactive, 0 stars, 5 downloads, Public
- gonzalors23 / gonzaloshop-buy-microservice**
Contains: Image | Last pushed: 12 days ago
Inactive, 0 stars, 4 downloads, Public
- gonzalors23 / gonzaloshop-frontend**
Contains: Image | Last pushed: 15 days ago
Inactive, 0 stars, 5 downloads, Public
- gonzalors23 / gonzaloshop-mongodb-database**
Contains: Image | Last pushed: 16 days ago
Inactive, 0 stars, 4 downloads, Public
- gonzalors23 / gonzaloshop-mysql-database**
Contains: Image | Last pushed: 17 days ago
Inactive, 0 stars, 5 downloads, Public
- gonzalors23 / gonzaloshop-graphql-service**
Contains: Image | Last pushed: 17 days ago
Inactive, 0 stars, 5 downloads, Public
- gonzalors23 / gonzaloshop-product-microservice**
Contains: Image | Last pushed: 17 days ago
Inactive, 0 stars, 5 downloads, Public

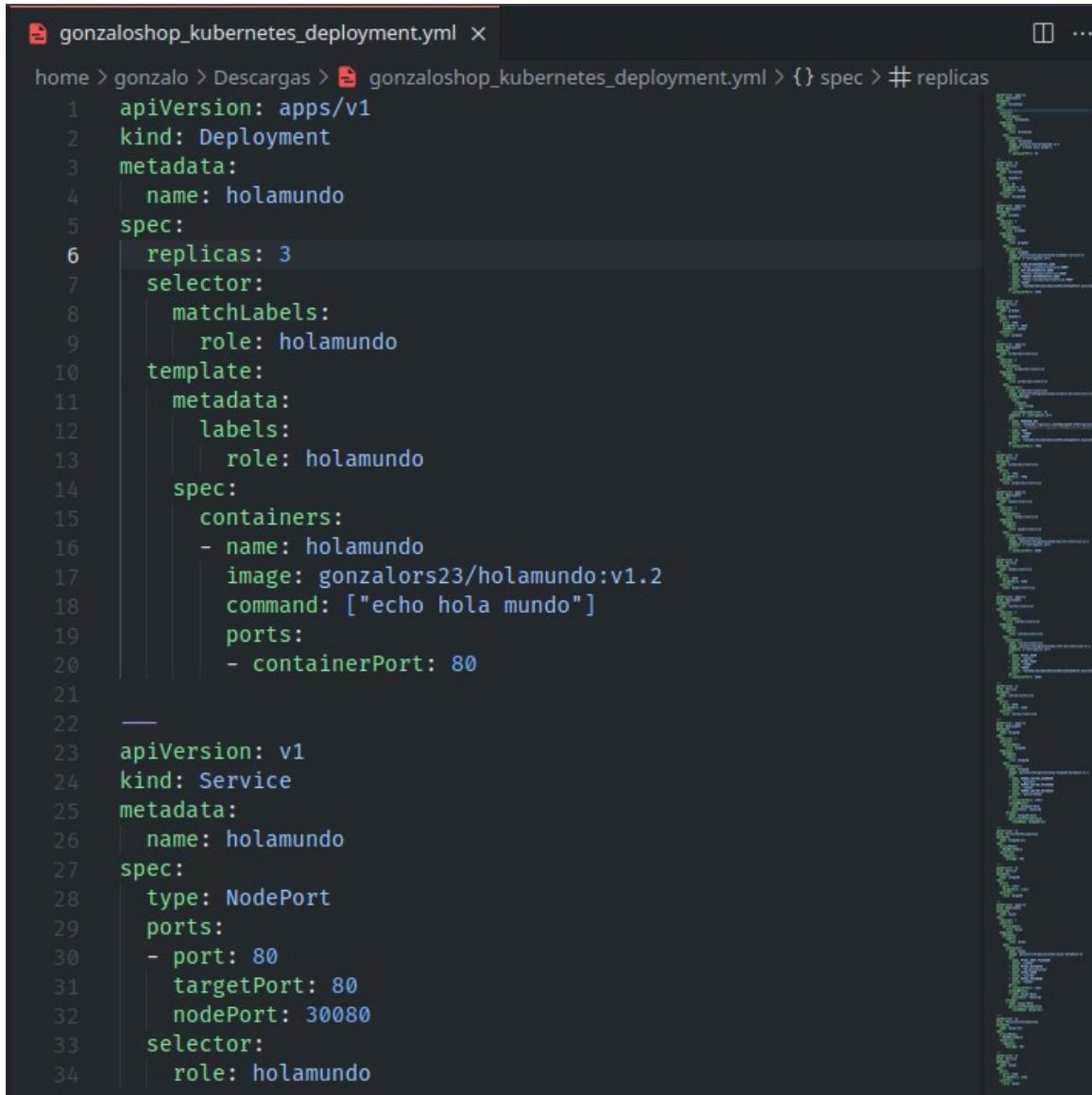
3.4.2-Creación de los archivos para el despliegue en Kubernetes.

Para desplegar nuestra aplicación con Kubernetes debemos crear un archivo donde especifiquemos toda la configuración sobre nuestro despliegue, pods, servicios y demás.

Para entender este archivo debemos conocer algunos conceptos sobre Kubernetes:

- Pod → Es el equivalente a un contenedor Docker.
- Deployments → Configuración que se aplica a un número de pods específicos.
- Services → Define un conjunto de pods y se encarga de dar la cara por ellos, recibir sus peticiones y repartir el trabajo entre todos sus pods.

Veamos un ejemplo:



The screenshot shows a code editor window with a dark theme. The file is named "gonzaloshop_kubernetes_deployment.yml". The code defines a Deployment and a Service. The Deployment section includes fields for apiVersion, kind, metadata (name: holamundo), spec (replicas: 3, selector, template, spec with containers and ports), and a long line separator. The Service section follows, with fields for apiVersion, kind, metadata (name: holamundo), spec (type: NodePort, ports, selector), and a long line separator. The code uses YAML syntax with numbered line numbers on the left.

```
home > gonzalo > Descargas > gonzaloshop_kubernetes_deployment.yml > {} spec > # replicas
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: holamundo
5  spec:
6    replicas: 3
7    selector:
8      matchLabels:
9        role: holamundo
10   template:
11     metadata:
12       labels:
13         role: holamundo
14     spec:
15       containers:
16         - name: holamundo
17           image: gonzalors23/holamundo:v1.2
18           command: ["echo hola mundo"]
19           ports:
20             - containerPort: 80
21
22
23   apiVersion: v1
24   kind: Service
25   metadata:
26     name: holamundo
27   spec:
28     type: NodePort
29     ports:
30       - port: 80
31         targetPort: 80
32         nodePort: 30080
33     selector:
34       role: holamundo
```

Como vemos tenemos un Deployment llamado holamundo, que corre la imagen de hola mundo y su función es mostrar el texto de “hola mundo” por consola ,esto sería lo mismo que correr con Docker la imagen gonzalors23/holamundo:v1.2. ¿Cuál es entonces la diferencia? Pues que en un Deployment podemos especificar un número de réplicas, en nuestro caso son tres.

Esto significa que nuestro Deployment tiene la función de crear tres pods exactamente iguales, pero si son iguales los tres, ¿cómo saben los otros pods a qué pod mandar una petición? Para ello están los servicios. El servicio siempre da la cara por sus pods, si yo quiero hacer una petición a un pod de hola mundo tendría que hacérsela al servicio holamundo, y este se encargaría de reenviar la petición al pod que considere oportuno.

De esta forma el trabajo se distribuye, ningún pod tiene una sobrecarga de trabajo y todo el despliegue en general está más balanceado y optimizado.

3.4.2.1-Deployment y Service de nuestro Frontend.

Para nuestro Frontend solo usamos una réplica de nuestra imagen del Frontend alojada en Docker Hub. Exponemos el servicio a través del puerto 30080.

```
gonzaloshop_kubernetes_deployment.yml x
home > gonzalo > Descargas > gonzaloshop_kubernetes_deployment.yml > ...
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: frontend
5  spec:
6    replicas: 1
7    selector:
8      matchLabels:
9        role: frontend
10   template:
11     metadata:
12       labels:
13         role: frontend
14     spec:
15       containers:
16         - name: frontend
17           image: gonzalors23/gonzaloshop-frontend:v1.2
18           command: ["./entrypoint.sh", "http://localhost:30500"]
19         ports:
20           - containerPort: 80
21
22   —
23   apiVersion: v1
24   kind: Service
25   metadata:
26     name: frontend
27   spec:
28     type: NodePort
29     ports:
30       - port: 80
31         targetPort: 80
32         nodePort: 30080
33     selector:
34       role: frontend
```

3.4.2.2-Deployment y Service de nuestro servidor GraphQL.

Para nuestro servidor GraphQL usamos dos réplicas de nuestra imagen alojada en Docker Hub, ya que es el componente de nuestra aplicación que más carga de trabajo tendrá. Debemos indicar las variables de entorno con las URLs de los demás microservicios

Exponemos el servicio a través del puerto 30500.

```
gonzaloshop_kubernetes_deployment.yml ×

home > gonzalo > Descargas > gonzaloshop_kubernetes_deployment.yml > {} spec > {} template > {} spec > [ ] contain
37   apiVersion: apps/v1
38   kind: Deployment
39   metadata:
40     name: graphql
41   spec:
42     replicas: 2
43     selector:
44       matchLabels:
45         role: graphql
46     template:
47       metadata:
48         labels:
49           role: graphql
50       spec:
51         containers:
52           - name: graphql
53             image: gonzalors23/gonzaloshop-graphql-service:v1
54             command: ["/entrypoint.sh"]
55             env:
56               - name: USER_MICROSERVICE_ADDR
57                 value: "http://usermicroservice:8000"
58               - name: BUY_MICROSERVICE_ADDR
59                 value: "http://buymicroservice:8000"
59               - name: PRODUCT_MICROSERVICE_ADDR
59                 value: "http://productmicroservice:7000"
59             ports:
60               - containerPort: 4000
61
62   —
63
64
65
66   apiVersion: v1
67   kind: Service
68   metadata:
69     name: graphql
70   spec:
71     type: NodePort
72     ports:
73       - port: 4000
74         targetPort: 4000
75         nodePort: 30500
76     selector:
77       role: graphql
```

3.4.2.3-Deployment y Service de nuestro microservicio de Productos.

Para nuestro microservicio de productos usamos dos réplicas de nuestra imagen alojada en Docker Hub, indicamos las variables de entorno con la URL de la base de datos MongoDB y el puerto donde se ejecutará la aplicación en la red interna de Kubernetes.

```
gonzaloshop_kubernetes_deployment.yml x
home > gonzalo > Descargas > gonzaloshop_kubernetes_deployment.yml > {} spec > {} template > {} spec > [ ] co
82  apiVersion: apps/v1
83  kind: Deployment
84  metadata:
85    name: productmicroservice
86  spec:
87    replicas: 2
88    selector:
89      matchLabels:
90        role: productmicroservice
91    template:
92      metadata:
93        labels:
94          role: productmicroservice
95    spec:
96      containers:
97        - name: productmicroservice
98          image: gonzalors23/gonzaloshop-product-microservice:v1
99          livenessProbe:
100            exec:
101              command:
102                - /bin/sleep
103                - "20"
104            initialDelaySeconds: 20
105            command: ["./entrypoint.sh"]
106            env:
107              - name: MONGODB_URI
108                value: "mongodb://gonzalo:12345@mongodb:27017/gonzaloShop"
109              - name: PORT
110                value: "7000"
111            ports:
112              - containerPort: 7000
113
114  —
115  apiVersion: v1
116  kind: Service
117  metadata:
118    name: productmicroservice
119  spec:
120    ports:
121      - port: 7000
122        targetPort: 7000
123    selector:
124      role: productmicroservice
```

3.4.2.4-Deployment y Service de nuestro microservicio de Usuarios.

Para nuestro microservicio de usuarios usamos dos réplicas de nuestra imagen alojada en Docker Hub, indicamos las variables de entorno para la conexión con la base de datos MySQL.

```
gonzaloshop_kubernetes_deployment.yml x
home > gonzalo > Descargas > gonzaloshop_kubernetes_deployment.yml > {} spec > {} template > {} spec >
161  apiVersion: apps/v1
162  kind: Deployment
163  metadata:
164    name: usermicroservice
165  spec:
166    replicas: 2
167    selector:
168      matchLabels:
169        role: usermicroservice
170    template:
171      metadata:
172        labels:
173          role: usermicroservice
174    spec:
175      containers:
176        - name: usermicroservice
177          image: gonzalors23/gonzaloshop-user-microservice:v1.2
178          command: ["/entrypoint.sh"]
179          env:
180            - name: MYSQL_ADDR
181              value: "mysql"
182            - name: MYSQL_PORT
183              value: "3306"
184          ports:
185            - containerPort: 8000
186
187  —
188  apiVersion: v1
189  kind: Service
190  metadata:
191    name: usermicroservice
192  spec:
193    ports:
194      - port: 8000
195        targetPort: 8000
196    selector:
197      role: usermicroservice
198
```

3.4.2.5-Deployment y Service de nuestro microservicio de Compras.

Para nuestro microservicio de usuarios usamos una réplica de nuestra imagen alojada en Docker Hub, no indicamos variables de entorno ya que estas vienen dentro del fichero env de Laravel.

```
gonzaloshop_kubernetes_deployment.yml x
home > gonzalo > Descargas > gonzaloshop_kubernetes_deployment.yml > {} spec > {} template > {} spec > [ ] containers
127  apiVersion: apps/v1
128  kind: Deployment
129  metadata:
130    name: buymicroservice
131  spec:
132    replicas: 1
133    selector:
134      matchLabels:
135        role: buymicroservice
136    template:
137      metadata:
138        labels:
139          role: buymicroservice
140    spec:
141      containers:
142        - name: buymicroservice
143          image: gonzalors23/gonzaloshop-buy-microservice:v1.3
144          command: ["/entrypoint.sh"]
145          ports:
146            - containerPort: 8000
147
148  —
149  apiVersion: v1
150  kind: Service
151  metadata:
152    name: buymicroservice
153  spec:
154    ports:
155      - port: 8000
156        targetPort: 8000
157    selector:
158      role: buymicroservice
159
```

3.4.2.6-Deployment, Service y Volumen de MongoDB.

Para las bases de datos además de el Deployment y el Service también especificamos un volumen. Un volumen nos permite guardar información de manera persistente ya que los pods y contenedores son volátiles, si no especificamos los volúmenes en nuestras bases de datos, aparecerían vacías cada vez que se iniciaran.

Para nuestra base de datos MongoDB usamos una réplica de nuestra imagen alojada en Docker Hub, debemos indicarle las variables de entorno con el usuario y contraseña para la base de datos y el nombre de la base de datos donde guardaremos nuestros productos.

```
gonzaloshop_kubernetes_deployment.yml ×
home > gonzalo > Descargas > gonzaloshop_kubernetes_deployment.yml > {} spec > {} template > {} spec >
200  apiVersion: apps/v1
201  kind: Deployment
202  metadata:
203    name: mongodb
204  spec:
205    replicas: 1
206    selector:
207      matchLabels:
208        role: mongodb
209    template:
210      metadata:
211        labels:
212          role: mongodb
213      spec:
214        containers:
215          - name: mongodb
216            image: gonzalors23/gonzaloshop-mongodb-database:v1.1
217            env:
218              - name: MONGO_INITDB_USERNAME
219                value: "gonzalo"
220              - name: MONGO_INITDB_PASSWORD
221                value: "12345"
222              - name: MONGO_INITDB_DATABASE
223                value: "gonzaloShop"
224            ports:
225              - containerPort: 27017
226            volumeMounts:
227              - name: mongodb-data
228                mountPath: /data/db
229            volumes:
230              - name: mongodb-data
231                persistentVolumeClaim:
232                  claimName: mongodb-pvc
233
```

También debemos especificar que volumen usará. Para la creación del volumen configuramos el nombre, el modo de acceso y la capacidad.

```
235  apiVersion: v1
236  kind: PersistentVolumeClaim
237  metadata:
238    name: mongodb-pvc
239  spec:
240    accessModes:
241      - ReadWriteOnce
242    resources:
243      requests:
244        storage: 5Gi
245
```

A parte claro está del servicio de MongoDB.

```
apiVersion: v1
kind: Service
metadata:
  name: mongodb
spec:
  ports:
    - port: 27017
      targetPort: 27017
  selector:
    role: mongodb
```

3.4.2.7-Deployment, Service y Volumen de MySQL.

Para nuestra base de datos MySQL usamos una réplica de nuestra imagen alojada en Docker Hub, debemos indicarle las variables de entorno con el usuario y contraseña para la base de datos, la contraseña del usuario root y el nombre de la base de datos donde guardaremos nuestros registros.

```
gonzaloshop_kubernetes_deployment.yml ×
home > gonzalo > Descargas > gonzaloshop_kubernetes_deployment.yml > {} spec > {} template > {} spec > [ ] containers > {} o
259  apiVersion: apps/v1
260  kind: Deployment
261  metadata:
262    name: mysql
263  spec:
264    replicas: 1
265    selector:
266      matchLabels:
267        role: mysql
268    template:
269      metadata:
270        labels:
271          role: mysql
272    spec:
273      containers:
274        - name: mysql
275          image: gonzalors23/gonzaloshop-mysql-database:v1
276          env:
277            - name: MYSQL_ROOT_PASSWORD
278              value: "12345"
279            - name: MYSQL_DATABASE
280              value: "db_gonzaloshop"
281            - name: MYSQL_USER
282              value: "userdb"
283            - name: MYSQL_PASSWORD
284              value: "12345"
285          ports:
286            - containerPort: 3306
287          volumeMounts:
288            - name: mysql-data
289              mountPath: /data/db
290        volumes:
291          - name: mysql-data
292            persistentVolumeClaim:
293              claimName: mysql-pvc
```

Volumen de MySQL:

```
295  —
296  apiVersion: v1
297  kind: PersistentVolumeClaim
298  metadata:
299    name: mysql-pvc
300  spec:
301    accessModes:
302      - ReadWriteOnce
303    resources:
304      requests:
305        storage: 5Gi
```

Servicio de MySQL:

```
308  apiVersion: v1
309  kind: Service
310  metadata:
311    name: mysql
312  spec:
313    ports:
314      - port: 3306
315        targetPort: 3306
316    selector:
317      role: mysql
```

CAPÍTULO 4: PRUEBAS DE LA APLICACIÓN

4.1-Introducción.

En este punto voy a hablar sobre el testing de la aplicación, en lugar de probar la aplicación manualmente he decidido realizar una batería de pruebas unitarias a cada microservicio y otra batería de pruebas End to End a todo el proyecto entero. A continuación veremos las pruebas de cada componente y los resultados que arrojan.

4.2-Realizando pruebas al microservicio de productos.

Para realizar los test al microservicio de productos usamos las librerías de Jest y Supertest, dos librerías de JavaScript que nos permiten realizar test unitarios a nuestros microservicios.

4.2.1-Test para pedir todos los productos.

En el primer test pedimos todos los productos, comprobamos que la respuesta sea un JSON, que devuelva un código 200 y que la respuesta tenga el campo name definido (si existe un campo name es que ha devuelto los productos).

```
test('Pedir todos los productos', async () => {
  const response = await api
    .get(PATHS_PRODUCTS.todos)
    .expect(200)
    .expect('Content-Type', /application\json/)

  expect(response.body[0].name).toBeDefined()
})
```

4.2.2-Test para pedir tres productos.

En el siguiente test pedimos tres productos para comprobar que el filtro de numero de productos funciona correctamente. Comprobamos que la respuesta sea un JSON, que devuelva un código 200 , que la respuesta tenga el campo name y solo devuelvan tres objetos.

```
test('Pedir 3 productos', async () => {
  const response = await api
    .get(PATHS_PRODUCTS.tresProductos)
    .expect(200)
    .expect('Content-Type', /application\json/)

  expect(response.body[0].name).toBeDefined()
  expect(response.body.length).toBe(3)
})
```

4.2.4-Test del buscador correcto.

En el siguiente test hacemos uso del buscador, buscando por “Disco duro”. Comprobamos que la respuesta sea un JSON, que devuelva un código 200 y que la respuesta tenga el campo name, solo devuelvan tres objetos ya que solo tenemos tres discos duros como productos.

```
test('Pruebas del buscador correcto', async () => {
  const response = await api
    .get(PATHS_PRODUCTS.buscarDiscoDuro)
    .expect(200)
    .expect('Content-Type', /application\json/)

  expect(response.body[0].name).toBeDefined()
  expect(response.body.length).toBe(3)
})
```

4.2.5-Test del buscador erróneo.

En el siguiente test hacemos uso del buscador intentando buscar un producto que no exista y comprobando que devuelva un error. Comprobamos que la respuesta sea un JSON y que la respuesta tenga el campo error definido.

```
test('Pruebas del buscador erroneo', async () => {
  const response = await api
    .get(PATHS_PRODUCTS.buscarProductoFalso)
    .expect('Content-Type', /application\json/)
  expect(response.body[0].error).toBeDefined()
})
```

4.2.6-Test productos con descuento.

En el siguiente test filtramos solo los productos que estén en oferta. Comprobamos que la respuesta sea un JSON, que la respuesta tenga el campo name definido y que el campo oferta sea true.

```
test('Pedir productos con descuento', async () => {
  const response = await api
    .get(PATHS_PRODUCTS.productosDescuentos)
    .expect(200)
    .expect('Content-Type', /application\json/)

  expect(response.body[0].name).toBeDefined()
  expect(response.body[0].sale).toBe(true)
})
```

4.2.7-Test filtrar por precio correcto.

En el siguiente test filtramos solo los productos cuyo precio esté entre los 50 y los 100 euros. Comprobamos que la respuesta sea un JSON, que mande un código 200, que la respuesta tenga el campo name definido y que el precio se encuentre entre 50 y 100.

```
test('Precio entre 50 y 100 €', async () => {
  const response = await api
    .get(PATHS_PRODUCTS.precioEntre50y100)
    .expect(200)
    .expect('Content-Type', /application\+json/)

  expect(response.body[0].name).toBeDefined()
  expect(response.body[0].price >= 50 && response.body[0].price <= 100)
})
```

4.2.8-Test filtrar por precio erróneo.

En el siguiente test filtramos los productos por precio pero usamos un precio mínimo más alto que el máximo por lo que nos muestra un mensaje de error. Comprobamos que la respuesta sea un JSON y que el campo error esté definido.

```
test('Precio entre 50 y 100 € erroneo', async () => {
  const response = await api
    .get(PATHS_PRODUCTS.precioErroneo)
    .expect('Content-Type', /application\+json/)
  expect(response.body[0].error).toBeDefined()
})
```

4.2.9-Test buscar un producto correcto.

En el siguiente test buscamos un producto que exista, para ello pedimos todos los productos y nos buscamos por el id del primero que aparezca. Comprobamos que la respuesta sea un JSON y que el nombre del producto buscado sea igual que el de la respuesta.

```
test('Pedir un producto correcto', async () => {
  const products = await api
    .get(PATHS_PRODUCTS.todos)

  const response = await api
    .get(`api/product/${products.body[0].id}`)
    .expect(200)
    .expect('Content-Type', /application\+json/)

  expect(response.body.name).toBe(products.body[0].name)
})
```

4.2.10-Test buscar un producto erróneo.

En el siguiente test buscamos un producto que no existe. Comprobamos que la respuesta sea un JSON, que el código de respuesta sea 500 y que el campo error esté definido.

```
test('Pedir un producto erroneo', async () => {
  const response = await api
    .get('/api/product/idfalso')
    .expect(500)
    .expect('Content-Type', /application\json/)

  expect(response.body.error).toBeDefined()
})
```

4.2.11-Test añadir una valoración.

En el siguiente test buscamos añadimos una valoración correctamente. Comprobamos que la respuesta sea un JSON, que el código de respuesta sea 200 y que la ultima valoración sea del usuario de prueba.

```
test('Añadir valoración correcta', async () => {
  const valoration = {
    username: "usuario_prueba",
    text: "",
    stars: 5
  }

  const products = await api
    .get(PATHS_PRODUCTS.todos)

  const response = await api
    .put(`/api/product/${products.body[0].id}`)
    .send(valoration)
    .expect(200)
    .expect('Content-Type', /application\json/)

  expect(response.body.valorations[-1].username).toBe("usuario_prueba")
})
```

4.2.12-Test añadir una valoración errónea.

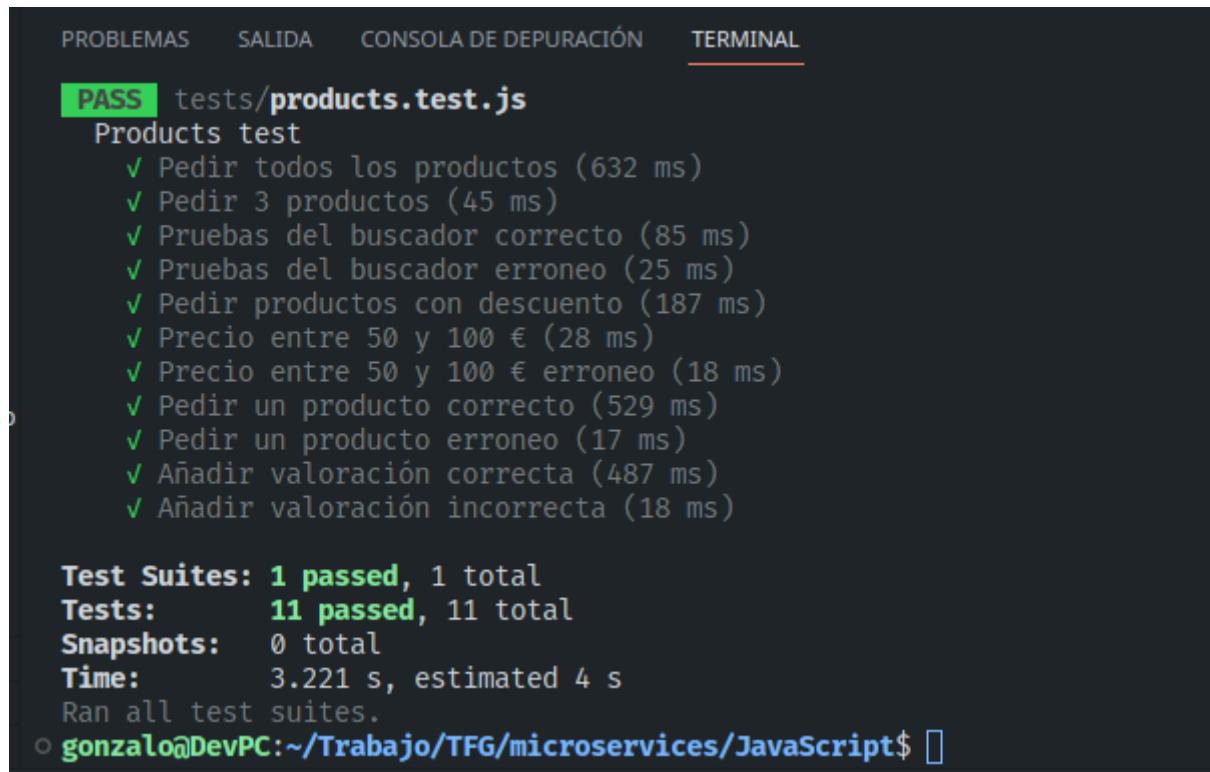
En el siguiente test buscamos añadimos una valoración errónea. Comprobamos que la respuesta sea un JSON y que el código de respuesta sea 500 .

```
test('Añadir valoración incorrecta', async () => {
  const valoration = {
    username: "usuario_prueba",
    text: "",
    stars: 5
  }

  await api
    .put('/api/product/noexiste')
    .send(valoration)
    .expect(500)
    .expect('Content-Type', /application\+json/)
})
```

4.2.13-Ejecución de los tests.

Como vemos los test se ejecutan correctamente:



```
PASS tests/products.test.js
Products test
  ✓ Pedir todos los productos (632 ms)
  ✓ Pedir 3 productos (45 ms)
  ✓ Pruebas del buscador correcto (85 ms)
  ✓ Pruebas del buscador erroneo (25 ms)
  ✓ Pedir productos con descuento (187 ms)
  ✓ Precio entre 50 y 100 € (28 ms)
  ✓ Precio entre 50 y 100 € erroneo (18 ms)
  ✓ Pedir un producto correcto (529 ms)
  ✓ Pedir un producto erroneo (17 ms)
  ✓ Añadir valoración correcta (487 ms)
  ✓ Añadir valoración incorrecta (18 ms)

Test Suites: 1 passed, 1 total
Tests:       11 passed, 11 total
Snapshots:   0 total
Time:        3.221 s, estimated 4 s
Ran all test suites.
gonzalo@DevPC:~/Trabajo/TFG/microservices/JavaScript$
```

4.3-Realizando pruebas al microservicio de usuarios.

Para realizar los test al microservicio de usuarios usamos las librerías de Django test y los test de rest_framework.

4.3.1-Test login correcto.

En el primer test realizamos un login correcto y comprobamos que el código de respuesta sea 201, esto significa que el usuario se ha creado correctamente.

```
def test_login_user(self):
    data = {
        'username': 'pruebaLogin',
        'password': '12345',
    }
    response = self.client.post(self.login_url, data, format='json')
    self.assertEqual(response.status_code, 201)
```

4.3.2-Test login con contraseña incorrecta.

En el siguiente test realizamos un login con una contraseña incorrecta y comprobamos que el código de respuesta sea 406, esto significa que el usuario existe pero la contraseña no es correcta.

```
def test_bad_password_login_user(self):
    data = {
        'username': 'pruebaLogin',
        'password': '12325',
    }
    response = self.client.post(self.login_url, data, format='json')
    self.assertEqual(response.status_code, 406)
```

4.3.3-Test login con usuario incorrecto.

En el siguiente test realizamos un login con un usuario que no existe y comprobamos que el código de respuesta sea 500, esto significa que el usuario no existe.

```
def test_bad_user_login_user(self):
    data = {
        'username': 'nonexist',
        'password': '12345',
    }
    response = self.client.post(self.login_url, data, format='json')
    self.assertEqual(response.status_code, 500)
```

4.3.4-Test creación de usuario correcta.

En el siguiente test creamos un usuario correctamente y comprobamos que el estado sea 201 lo que significa que el usuario se ha creado correctamente.

```
def test_post_user(self):
    data = {
        'name': 'prueba',
        'lastname': 'prueba',
        'username': 'prueba',
        'email': 'email@prueba.com',
        'password': '12345',
        'bank_account': '12345678901234567890'
    }
    response = self.client.post(self.usuarios_url, data, format='json')
    self.assertEqual(response.status_code, 201)
```

4.3.5-Test creación de usuario con username erróneo.

En el siguiente test creamos un usuario con un username erróneo y comprobamos que el código de error sea 411.

```
def test_bad_user_post_user(self):
    databad = {
        'name': 'prueba',
        'lastname': 'prueba',
        'username': 'as',
        'email': 'email@prueba.com',
        'password': 'prueba',
        'bank_account': '12345678901234567890'
    }
    response = self.client.post(self.usuarios_url, databad, format='json')
    self.assertEqual(response.status_code, 411)
```

4.3.6-Test creación de usuario con contraseña errónea.

En el siguiente test creamos un usuario con una contraseña errónea y comprobamos que el código de error sea 411.

```
def test_bad_password_post_user(self):
    databad = {
        'name': 'prueba',
        'lastname': 'prueba',
        'username': 'prueba',
        'email': 'email@prueba.com',
        'password': 'as',
        'bank_account': '12345678901234567890'
    }
    response = self.client.post(self.usuarios_url, databad, format='json')
    self.assertEqual(response.status_code, 411)
```

4.3.7-Test creación de usuario con número de cuenta erróneo.

En el siguiente test creamos un usuario con un número de cuenta erróneo y comprobamos que el código de error sea 411.

```
def test_bad_bank_account_post_user(self):
    databad = {
        'name': 'prueba',
        'lastname': 'prueba',
        'username': 'prueba',
        'email': 'email@prueba.com',
        'password': 'prueba',
        'bank_account': '678901234567890'
    }
    response = self.client.post(self.usuarios_url, databad, format='json')
    self.assertEqual(response.status_code, 411)
```

4.3.8-Test creación de usuario con username repetido.

En el siguiente test creamos un usuario con un username en uso, para ello creamos un usuario y seguidamente creamos otro con el mismo username. Comprobamos que el código de error sea 500.

```
def test_repeat_username_post_user(self):
    data1 = {
        'name': 'prueba',
        'lastname': 'prueba',
        'username': 'prueba',
        'email': 'email@prueba.com',
        'password': 'prueba',
        'bank_account': '12345678901234567890'
    }
    data2 = {
        'name': 'prueba',
        'lastname': 'prueba',
        'username': 'prueba',
        'email': 'email@prueba2.com',
        'password': 'prueba',
        'bank_account': '12345678901234567890'
    }
    self.client.post(self.usuarios_url, data1, format='json')
    response = self.client.post(self.usuarios_url, data2, format='json')
    self.assertEqual(response.status_code, 500)
```

4.3.9-Test creación de usuario con email repetido.

En el siguiente test creamos un usuario con un email en uso, para ello creamos un usuario y seguidamente creamos otro con el mismo email. Comprobamos que el código de error sea 500.

```
def test_repeat_email_post_user(self):
    data1 = {
        'name': 'prueba',
        'lastname': 'prueba',
        'username': 'prueba1',
        'email': 'email@prueba.com',
        'password': 'prueba',
        'bank_account': '12345678901234567890'
    }
    data2 = {
        'name': 'prueba',
        'lastname': 'prueba',
        'username': 'prueba2',
        'email': 'email@prueba.com',
        'password': 'prueba',
        'bank_account': '12345678901234567890'
    }
    self.client.post(self.usuarios_url, data1, format='json')
    response = self.client.post(self.usuarios_url, data2, format='json')
    self.assertEqual(response.status_code, 500)
```

4.3.10-Test añadir dinero al monedero.

En el siguiente test añadimos 10 euros al monedero del usuario de prueba. Comprobamos que el código de estado sea 200.

```
def test_add_money_user(self):
    data = {
        'username': 'prueba',
        'money': 10,
    }
    response = self.client.put(self.usuarios_url, data, format='json')
    self.assertEqual(response.status_code, 200)
```

4.3.11-Test añadir dinero al monedero de un usuario inexistente.

En el siguiente test añadimos 10 euros al monedero de un usuario que no existe. Comprobamos que nos devuelve un error con código de estado sea 500.

```
def test_bad_add_money_user(self):
    data = {
        'username': 'nonexist',
        'money': 10,
    }
    response = self.client.put(self.usuarios_url, data, format='json')
    self.assertEqual(response.status_code, 500)
```

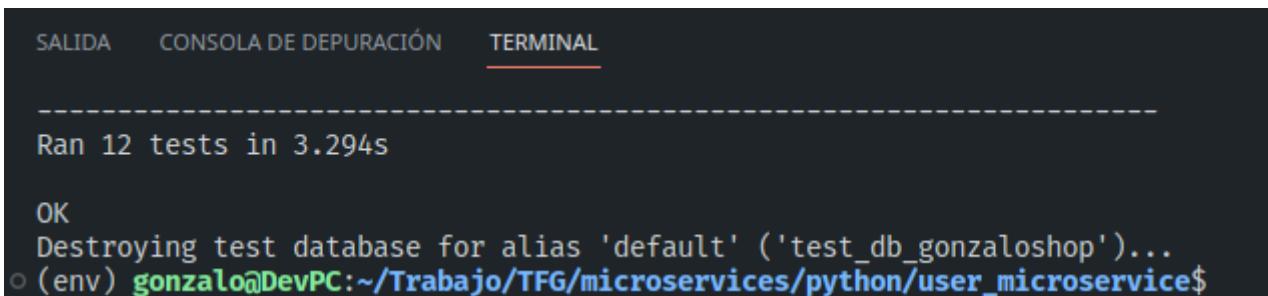
4.3.12-Test añadir dinero por encima del límite.

En el siguiente test añadimos 1001 euros al monedero del usuario de prueba cuyo límite es 1000. Comprobamos que nos devuelve un error con código de estado sea 500.

```
def test_add_much_money_user(self):
    data = {
        'username': 'prueba',
        'money': 1001,
    }
    response = self.client.put(self.usuarios_url, data, format='json')
    self.assertEqual(response.status_code, 500)
```

4.3.13-Ejecución de los test.

Como vemos los test se ejecutan correctamente:



The screenshot shows a terminal window with three tabs at the top: SALIDA, CONSOLA DE DEPURACIÓN, and TERMINAL. The TERMINAL tab is selected and contains the following output:

```
-->
Ran 12 tests in 3.294s
OK
Destroying test database for alias 'default' ('test_db_gonzaloshop')...
o (env) gonzalo@DevPC:~/Trabajo/TFG/microservices/python/user_microservice$
```

4.4-Realizando pruebas al microservicio de compras.

Para realizar los test al microservicio de compras usamos las librerías de Laravel test, una librería que se incluye dentro del framework y nos permite realizar test unitarios.

4.4.1-Test realizar una compra.

En el primer test realizamos una compra y comprobamos que el código de respuesta sea 200, esto significa que la compra se ha realizado correctamente.

```
public function test_crate_compra(){
    $articulosPedido = array();
    $articulo1 = array(
        "precio" => 10,
        "nombre" => "articulo de prueba",
        "cantidad" => 1
    );

    array_push($articulosPedido, $articulo1);

    $response = $this->post("/api/compras", [
        "idUsuario" => 1,
        "precioTotal" => 10,
        "fechaPedido" => "2023-01-01",
        "fechaEntrega" => "2023-01-20",
        "articulos" => $articulosPedido
    ], ['HTTP_Authorization' => "tokenCorrecto"]);

    $response->assertStatus(200);
}
```

4.4.2-Test realizar una compra sin mandar token.

En el siguiente test realizamos una compra pero no especificamos el token. Comprobamos que el código de respuesta sea 401.

```

public function test_crate_compra_token_error(){
    $articulosPedido = array();
    $articulo1 = array(
        "precio" => 10,
        "nombre" => "articulo de prueba",
        "cantidad" => 1
    );

    array_push($articulosPedido, $articulo1);

    $response = $this→post("/api/compras", [
        "idUsuario" => 1,
        "precioTotal" => 10,
        "fechaPedido" => "2023-01-01",
        "fechaEntrega" => "2023-01-20",
        "articulos" => $articulosPedido
    ]);

    $response→assertStatus(401);
}

```

4.4.3-Test realizar una compra con usuario inexistente.

En el siguiente test realizamos una compra con un usuario inexistente. Comprobamos que el código de respuesta sea 401.

```

public function test_crate_compra_user_error(){
    $articulosPedido = array();
    $articulo1 = array(
        "precio" => 10,
        "nombre" => "articulo de prueba",
        "cantidad" => 1
    );

    array_push($articulosPedido, $articulo1);

    $response = $this→post("/api/compras", [
        "idUsuario" => 1523,
        "precioTotal" => 10,
        "fechaPedido" => "2023-01-01",
        "fechaEntrega" => "2023-01-20",
        "articulos" => $articulosPedido
    ],['HTTP_Authorization' => "tokenCorrecto"]);

    $response→assertStatus(401);
}

```

4.4.4-Test conseguir todas las compras del usuario 1.

En el siguiente test consultamos todas las compras del usuario con id 1 y comprobamos que el código de estado sea 200.

```
public function test_get_compras(){
    $response = $this→get(
        "/api/compras/1",
        ['HTTP_Authorization' => "tokenCorrecto"]
    );

    $response→assertStatus(200);
}
```

4.4.5-Test conseguir compras sin mandar token.

En el siguiente test consultamos todas las compras del usuario con id 1 pero no mandamos un token. Comprobamos que el código de error sea 401

```
public function test_get_compras_token_error(){
    $response = $this→get(
        "/api/compras/1"
    );

    $response→assertStatus(401);
}
```

4.4.6-Test conseguir compras de un usuario inexistente.

En el siguiente test consultamos todas las compras de un usuario inexistente. Comprobamos que el código de error sea 401

```
public function test_get_compras_user_error(){
    $response = $this→get(
        "/api/compras/1234",
        ['HTTP_Authorization' => "tokenCorrecto"]
    );

    $response→assertStatus(401);
}
```

4.4.7-Ejecución de los test.

Como vemos los test se ejecutan correctamente.

```
SALIDA CONSOLA DE DEPURACIÓN TERMINAL

● gonzalo@DevPC:~/Trabajo/TFG/microservices/PHP/buy_microservice$ php artisan test

  PASS Tests\Unit\ExampleTest
    ✓ that true is true

  PASS Tests\Feature\ComprasTest
    ✓ crate compra
    ✓ crate compra token error
    ✓ crate compra user error
    ✓ get compras
    ✓ get compras token error
    ✓ get compras user error

  PASS Tests\Feature\ExampleTest
    ✓ the application returns a successful response

Tests: 8 passed
Time: 0.72s

○ gonzalo@DevPC:~/Trabajo/TFG/microservices/PHP/buy_microservice$ █
```

4.5-Realizando pruebas al servidor GraphQL.

Para realizar los test al servidor GraphQL usamos las librerías de jest y supertest, librerías que nos permiten realizar test unitarios.

Para atacar con consultas GraphQL con nuestro servidor he creado una función que lanza la consulta con una petición POST. (recordemos que así funciona GraphQL).

```
const url = "http://127.0.0.1:4000/graphql/"

function makeQuery(query){
  const request = axios.post(url, query)
  return request.then(response => response.data)
}
```

4.5.1-Test microservicio de usuarios.

En el primer test comprobamos el correcto funcionamiento del microservicio de usuarios, creando un usuario correctamente y eliminándolo después. Comprobamos que el mensaje “eliminado correctamente” esté definido.

```
test('Microservicio Python funciona correctamente', async () => {
  const query1 = {
    "query": "mutation { createUser(name:\"usuarioPruebas\", lastname:\"usuario\", username:\"usuarioPruebas\", password:\"123456\") }"
  }

  const response1 = await makeQuery(query1)
  expect(response1.data.createUser.token).toBeDefined()

  const query2 = {
    "query": "mutation { delUser(token:${response1.data.createUser.token}){ __typename ... on Message {msg} ... on Error {error} } }"
  }

  const response2 = await makeQuery(query2)
  expect(response2.data.delUser.msg).toBeDefined()
})
```

4.5.2-Test errores del microservicio de usuarios.

En el siguiente test comprobamos el correcto manejo de errores del microservicio de usuarios eliminando un usuario que no existe.

```
test('Microservicio Python devuelve errores correctamente', async () => {
  const query = {
    "query": "mutation { delUser(token:\"lkjadsfoijeld\"){ __typename ... on Message {msg} ... on Error {error} } }"
  }

  const response = await makeQuery(query)
  expect(response.data.delUser.error).toBeDefined()
})
```

4.5.3-Test microservicio de productos.

En el siguiente test comprobamos el correcto funcionamiento del microservicio de productos, añadiendo una valoración a un producto. Comprobamos que el campo nombre aparezca en la valoración.

```
test('Microservicio JavaScript funciona correctamente', async () => {
  const query = {
    "query": "mutation { addValoration(ident: \"6447b73c262de0cf3a66167c\", username: \"manoloLama\", text: \"esta muy bien\") { name } }"
  }

  const response = await makeQuery(query)
  expect(response.data.addValoration.name).toBeDefined()
})
```

4.5.4-Test errores del microservicio de productos.

En el siguiente test comprobamos el correcto manejo de errores del microservicio de productos añadiendo una valoración a un producto que no existe. Comprobamos que el campo error esté definido.

```
test('Microservicio de JavaScript devuelve errores correctamente', async () => {
  const query = {
    "query": "mutation { addValoration(ident: \"noexisteelproducto\", username: \"manoloLama\", text: \"esta muy bien\") { error } }"
  }

  const response = await makeQuery(query)
  expect(response.data.addValoration.error).toBeDefined()
})
```

4.5.5-Test microservicio de compras.

En el siguiente test comprobamos el correcto funcionamiento del microservicio de compras, consultando las compras de un usuario. Comprobamos que el campo id esté definido.

```
test('Microservicio PHP funciona correctamente', async () => {
  const query = {
    "query": "mutation { getBuy(idUsuario: 1, token: \"tokentotalmentevalido\"){ __typename } }"
  }

  const response = await makeQuery(query)
  expect(response.data.getBuy[0].id).toBeDefined()
})
```

4.5.6-Test errores del microservicio de compras.

En el siguiente test comprobamos el correcto manejo de errores del microservicio de compras consultando las compras de un usuario que no existe. Comprobamos que el campo error esté definido.

```
test('Microservicio de PHP devuelve errores correctamente', async () => {
  const query = {
    "query": "mutation{getBuy(idUsuario: 1234, token: \"aldskffdgjoiejlka\"){ __typename ... on C
  }
  const response = await makeQuery(query)
  console.log(response)
  expect(response.errors).toBeDefined()
})
```

4.5.7-Ejecución de los test.

Como vemos los test se ejecutan correctamente.

```
PASS test/server.test.js
Server test
  ✓ Microservicio Python funciona correctamente (668 ms)
  ✓ Microservicio Python devuelve errores correctamente (26 ms)
  ✓ Microservicio JavaScript funciona correctamente (116 ms)
  ✓ Microservicio de JavaScript devuelve errores correctamente (16 ms)
  ✓ Microservicio PHP funciona correctamente (568 ms)
  ✓ Microservicio de PHP devuelve errores correctamente (127 ms)

Test Suites: 1 passed, 1 total
Tests:       6 passed, 6 total
Snapshots:  0 total
Time:        2.671 s
Ran all test suites.
```

4.6-Realizando pruebas e2e desde el Frontend.

Las pruebas e2e son pruebas que ejecutan la aplicación entera y simulan las acciones que un usuario normal tendría por la misma. Las pruebas constan de órdenes programadas como: pulsar un botón, escribir un texto en un formulario, navegar a la página tal, etc.

Al ejecutar una prueba e2e puedes ver como las acciones programadas se van ejecutando en tu aplicación y en el caso de error, como ha reaccionado.

Para realizar las pruebas he usado la librería de Cypress.

4.6.1-Testing creación de usuarios.

La primera parte que testearemos será la creación de usuarios, realizaremos los siguientes test:

4.6.1.1-Comprobación del menú nologin.

Comprobamos que en nuestra aplicación aparece el menú para loguearte en caso de que no lo estés. Para ello en el home de nuestra aplicación pulsamos el botón de identificarte y comprobamos que contenga la palabra inicia (de inicia sesión).

```
it('El menú nologin funciona correctamente', () => {
    cy.visit('http://localhost:3000')
    cy.get('#identify').click()
    cy.contains('Inicia')
})
```

4.6.1.2-Comprobación del formulario de registro.

Comprobamos que en nuestra aplicación aparece el formulario de registro dentro del apartado para registrarse. Para ello revisamos que el título del formulario “Crear cuenta” aparezca en la pantalla.

```
it('Aparece el formulario de registro', () => {
    cy.contains('Crear cuenta')
})
```

4.6.1.3-Comprobación error “campos obligatorios” formulario de registro.

Comprobamos que al intentar crear un usuario sin llenar los campos la aplicación nos arroja un error. Para ello pulsamos el botón de crear usuario y comprobamos que el texto “Todos los campos son obligatorios” aparezca en la pantalla.

```
it('Error campos sin llenar', () => {
    cy.get('#createUserButton').click()
    cy.contains('Todos los campos son obligatorios')
})
```

4.6.1.4-Comprobación error “contraseñas diferentes” formulario de registro.

Comprobamos que al intentar crear un usuario con una contraseña distinta en los campos “contraseña” y “repita su contraseña” del formulario, la aplicación nos arroja un error. Para ello escribimos en los dos campos de texto y comprobamos que “Las contraseñas no coinciden” aparezca en la pantalla.

```
it('Error contraseñas diferentes', () => {
    cy.get('#password1').type('contraseña1')
    cy.get('#password2').type('contraseña2')
    cy.contains('Las contraseñas no coinciden')
})
```

4.6.1.5-Comprobación error “usuario en uso” formulario de registro.

Comprobamos que al intentar crear un usuario con un nombre de usuario en uso la aplicación nos arroja un error. Para ello rellenamos el formulario y comprobamos que el texto “El nombre de usuario está en uso” aparezca en la pantalla.

```
it('Error usuario en uso', () => {
    cy.get('#nameInput').type('gonzalo')
    cy.get('#lastNameInput').type('apellido1')
    cy.get('#emailInput').type('correo@prueba.es')
    cy.get('#bankInput').type('121212121212121212')
    cy.get('#usernameInput').type('gonzalo')
    cy.get('#password1').type('12345')
    cy.get('#password2').type('12345')
    cy.get('#createUserButton').click()
    cy.contains('El nombre de usuario está en uso')
})
```

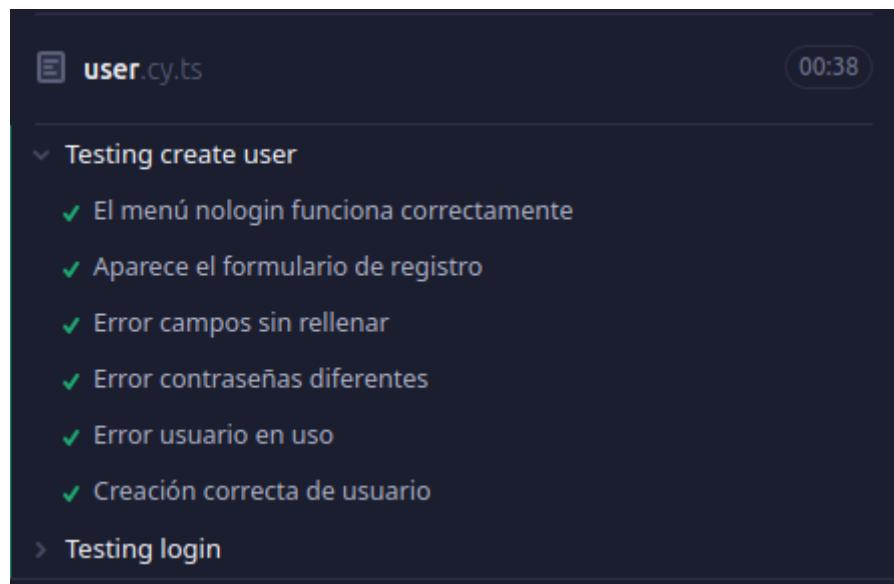
4.6.1.6-Comprobación creación de usuario correcta.

Creamos un usuario correctamente y comprobamos que el texto “creado correctamente” aparezca en la pantalla.

```
it('Creación correcta de usuario', () => {
    cy.get('#nameInput').type('nombre1')
    cy.get('#lastNameInput').type('apellido1')
    cy.get('#emailInput').type('correo@prueba.es')
    cy.get('#bankInput').type('121212121212121212')
    cy.get('#usernameInput').type('usuarioPrueba1')
    cy.get('#password1').type('12345')
    cy.get('#password2').type('12345')
    cy.get('#createUserButton').click()
    cy.contains('creado con éxito')
})
```

4.6.1.7-Ejecución de los test.

Ejecutamos los test de creación de usuarios.



4.6.2-Testing del login de usuarios.

La siguiente parte que testearemos será el login de usuarios, para ello realizaremos los siguientes test.

4.6.2.1-Comprobación error “contraseña incorrecta” login.

En el formulario de login nos logueamos con un usuario existente y una contraseña incorrecta. Comprobamos que nos aparece el error “Las contraseñas no coinciden”.

```
it('Error contraseña incorrecta', () => {
    cy.get('#identify').click()
    cy.contains('Inicia')
    cy.get('#usernameLogin').type('usuarioPrueba1')
    cy.get('#passwordLogin').type('contraseñaFalsa')
    cy.get('#loginButton').click()
    cy.contains('Las contraseñas no coinciden')
})
```

4.6.2.2-Comprobación error “usuario incorrecto” login.

En el formulario de login nos logueamos con un usuario inexistente. Comprobamos que nos aparece el error “Usuario no existente”.

```
it('Error usuario incorrecto', () => {
    cy.get('#identify').click()
    cy.contains('Inicia')
    cy.get('#usernameLogin').type('usuarioFalso')
    cy.get('#passwordLogin').type('12345')
    cy.get('#loginButton').click()
    cy.contains('Usuario no existente')
})
```

4.6.2.3-Comprobación login correcto.

En el formulario de login nos logueamos con un usuario y contraseña correctos. Comprobamos que nos aparece el mensaje “Sesión iniciada con éxito”.

```
it('Login correcto', () => {
    cy.get('#identify').click()
    cy.contains('Inicia')
    cy.get('#usernameLogin').type('usuarioPrueba1')
    cy.get('#passwordLogin').type('12345')
    cy.get('#loginButton').click()
    cy.contains('Sesión iniciada con éxito')
})
```

4.6.2.4-Comprobación deslogueo correcto.

Cerramos sesión con nuestro usuario y comprobamos que nos aparece el mensaje “Sesión cerrada correctamente”.

```
it('Deslogueo correcto', () => {
  cy.get('#identify').click()
  cy.contains('Inicia')
  cy.get('#usernameLogin').type('usuarioPrueba1')
  cy.get('#passwordLogin').type('12345')
  cy.get('#loginButton').click()
  cy.wait(6000)
  cy.get('#userMenu').click()
  cy.get('#unlogButton').click()
  cy.contains('Sesión cerrada correctamente')
})
```

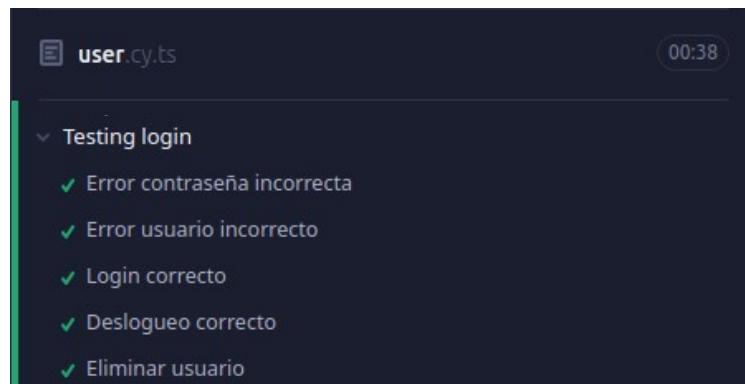
4.6.2.5-Comprobación eliminar usuario.

Vamos a eliminar el usuario creado anteriormente para realizar las pruebas de login. Para ello realizamos la acción desde la ventana emergente y comprobamos que nos aparece el mensaje de “usuario eliminado”.

```
it('Eliminar usuario', () => {
  cy.get('#identify').click()
  cy.contains('Inicia')
  cy.get('#usernameLogin').type('usuarioPrueba1')
  cy.get('#passwordLogin').type('12345')
  cy.get('#loginButton').click()
  cy.wait(6000)
  cy.get('#userMenu').click()
  cy.get('#delUserButton').click()
  cy.get('#confirmDelUser').click()
  cy.contains('Usuario eliminado')
})
```

4.6.2.6-Ejecución de los test.

Ejecutamos los test del login.



4.6.3-Testing del los productos.

En la siguiente parte testearemos la interacción con los productos, para ello realizaremos los siguientes test.

4.6.3.1-Comprobación error “no registrado” realizar valoración.

Añadimos una valoración sin iniciar sesión y comprobamos que nos aparece el error “Necesitas estar registrado”.

```
it('Añadir valoración error', () => {
  cy.get('#addValorationButton').click()
  cy.get('#sendValoración').click()
  cy.contains('Necesitas estar registrado')
})
```

4.6.3.2-Comprobación realizar valoración.

Añadimos una valoración correctamente y comprobamos que nos aparece el texto “Añadida correctamente”.

```
it('Añadir valoración correctamente', () => {
  cy.get('#identify').click()
  cy.contains('Inicia')
  cy.get('#usernameLogin').type('usuarioPrueba1')
  cy.get('#passwordLogin').type('12345')
  cy.get('#loginButton').click()
  cy.contains('Sesión iniciada con éxito')
  cy.wait(6000)
  cy.get('#addValorationButton').click()
  cy.get('#sendValoración').click()
  cy.contains('añadida correctamente')
})
```

4.6.3.3-Comprobación añadir producto al carrito.

Añadimos un producto al carrito correctamente y comprobamos que nos aparece el texto “Añadido”.

```
it('Añadir producto al carrito', () => {
  cy.get('#identify').click()
  cy.contains('Inicia')
  cy.get('#usernameLogin').type('usuarioPrueba1')
  cy.get('#passwordLogin').type('12345')
  cy.get('#loginButton').click()
  cy.contains('Sesión iniciada con éxito')
  cy.wait(6000)
  cy.get('#addToCartButton').click()
  cy.contains('añadido')
})
```

4.6.3.4-Comprobación comprar el producto.

Realizamos la compra del producto añadido correctamente y comprobamos que nos aparece el texto “Compra realizada”.

```
it('Comprar producto', () => {
    cy.get('#identify').click()
    cy.contains('Inicia')
    cy.get('#usernameLogin').type('usuarioPrueba1')
    cy.get('#passwordLogin').type('12345')
    cy.get('#loginButton').click()
    cy.contains('Sesión iniciada con éxito')
    cy.wait(6000)
    cy.visit('http://localhost:3000/cart')
    cy.get('#direction').type('dirección falsa para prueba')
    cy.get('#confirmBuy').click()
    cy.contains('Compra realizada')
})
```

4.6.3.5-Comprobación ver estado del envío.

Nos dirigimos a la pantalla donde vemos las compras del usuario y comprobamos que la tabla donde se muestran los envíos contenga el número uno, esto significa que la tabla aparece y solo tiene un articulo ya que este usuario se crea para realizar estas pruebas y no ha podido realizar mas compras.

```
it('Ver estado de envío', () => {
    cy.get('#identify').click()
    cy.contains('Inicia')
    cy.get('#usernameLogin').type('usuarioPrueba1')
    cy.get('#passwordLogin').type('12345')
    cy.get('#loginButton').click()
    cy.contains('Sesión iniciada con éxito')
    cy.wait(6000)
    cy.visit('http://localhost:3000/compras')
    cy.contains('1')
})
```

4.6.3.6-Ejecución de los test.

Ejecutamos los test de los productos.

```
product.cy.ts
00:55

> Preparación del usuario
└ Interacción con productos
  ✓ Añadir valoración error
  ✓ Añadir valoración correctamente
  ✓ Añadir producto al carrito
  ✓ Comprar producto
  ✓ Ver estado de envío
  ✓ Eliminar usuario
```

4.6.4-Testing de navegación.

En la siguiente parte testearemos la navegación en la aplicación, para ello realizaremos los siguientes test.

4.6.4.1-Comprobación del home.

Comprobamos que el home carga correctamente y aparecen los productos.

```
describe('Testing del home', () => {
  beforeEach(function() {
    cy.visit('http://localhost:3001')
  })

  it('El home carga correctamente', () => {
    cy.contains('GonzaloShop')
  })

  it('Aparece el apartado de productos', () => {
    cy.contains('Ofertas')
  })
})
```

4.6.4.2-Comprobación del apartado de productos.

Comprobamos que el apartado de productos carga correctamente.

```
it('Aparecen los productos', () => {
  cy.wait(1000)
  cy.contains('Sudadera')
})

it('Aparecen los filtros', () => {
  cy.contains('Filtros')
})
```

4.6.4.3-Comprobación de los filtros.

Filtramos por ropa y comprobamos que aparezcan sudaderas.

```
it('Funcionan los filtros', () => {
  cy.get('#category').select('Ropa')
  cy.get('#applyFilters').click()
  cy.wait(1000)
  cy.contains('Sudadera')
})
```

4.6.4.4-Comprobación del buscador.

Filtramos por disco y comprobamos que aparezcan los discos duros.

```
it('Funciona el buscador', () => {
  cy.get('#searchInput').type('disco')
  cy.get('#searchButton').click()
  cy.wait(1000)
  cy.contains('Disco')
})

it('Filtrar por disco', () => {
  cy.visit('http://localhost:3001/products/disco')
  cy.wait(1000)
  cy.contains('Kingston')
})
```

4.6.4.5-Comprobación único producto.

Visualizamos un producto específico y comprobamos que se renderiza correctamente.

```
it('Entrar en un producto', () => {
  cy.visit('http://localhost:3001/product/6447b73c262de0cf3a66166e')
  cy.wait(1000)
  cy.contains('Kingston')
  cy.contains('Capacidad')
  cy.contains('Valoraciones')
})
```

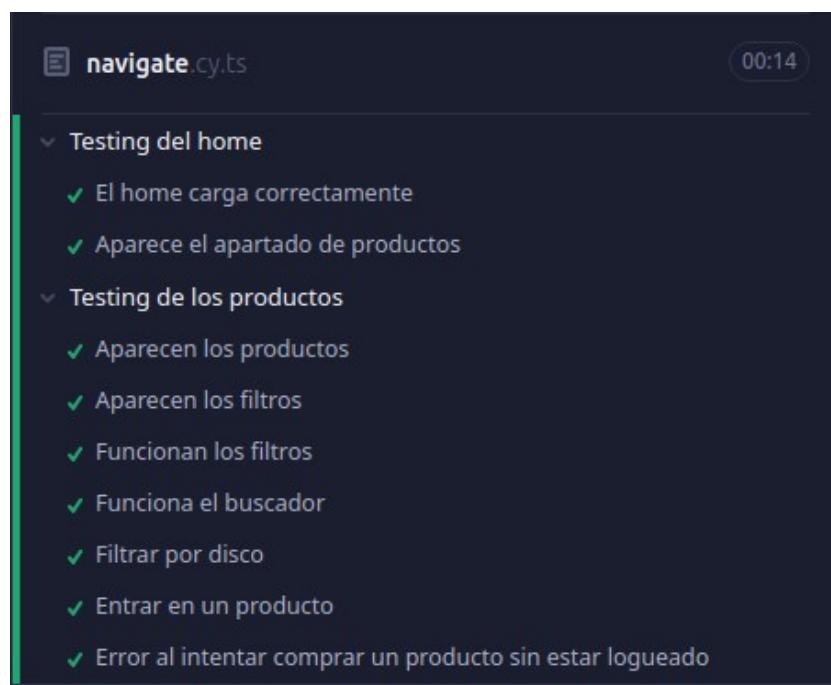
4.6.4.6-Comprobación error “no logueado” al comprar.

Intentamos realizar una compra sin estar logueados y comprobamos que aparece el error.

```
it('Error al intentar comprar un producto sin estar logueado', () => {
  cy.visit('http://localhost:3001/product/6447b73c262de0cf3a66166e')
  cy.wait(1000)
  cy.contains('Kingston')
  cy.get('#addToCartButton').click()
  cy.contains('Necesitas estar')
})
```

4.6.4.7-Ejecución de los test.

Los test se ejecutan correctamente.



CAPÍTULO 5: CONCLUSIONES

5.1-Conclusiones sobre el proyecto.

Realicé este proyecto para explorar las ventajas de los microservicios y exprimir al máximo su potencial, además de poder aprender Kubernetes y realizar mi primer despliegue de una aplicación en producción en la nube.

Realizando este proyecto he podido comprobar que el uso de los microservicios nos ofrece una gran cantidad de ventajas, como que te permite escalar y mantener cada microservicio de manera independiente y sin que las alteraciones en uno afecten a el resto de ellos, si ocurre un problema en una parte de la aplicación es solo esta parte la que deja de dar servicio y el resto del programa funciona correctamente, nos ofrece la posibilidad de utilizar tecnologías completamente diferentes y de manera transparente al resto y nos permite la creación de sistemas más robustos. No obstante, el uso de una arquitectura de microservicios agrega bastante complejidad a la aplicación.

También he podido comprobar que el uso de Kubernetes me ha ofrecido muchas ventajas en el despliegue de la aplicación, como una gran eficiencia a la hora de desplegar, una capacidad de escalar la aplicación y gestionar los pods increíble, es una tecnología de sencillo uso y simplifica mucho el trabajo necesario para realizar el despliegue de microservicios tanto localmente como en la nube.

La mezcla de todo esto me ha permitido crear una aplicación altamente escalable y con una tolerancia a fallos muy grande a parte de una reutilización de código importante.

5.2-Conclusiones finales.

Realizar este proyecto no ha sido nada fácil, ha costado muchas horas de esfuerzo y estudio. En cuanto al desarrollo quise plasmar todos mis conocimientos sobre la ingeniería de software y aunque ya contaba con grandes nociones sobre la mayoría de lenguajes y tecnologías que he usado para el desarrollo de la aplicación, conseguir el funcionamiento actual de la aplicación y aprender Kubernetes y realizar un despliegue de estas dimensiones ha sido un trabajo duro.

Estoy muy contento con el resultado y pienso que en el proyecto queda reflejado todo el esfuerzo realizado.

CAPÍTULO 6: BIBLIOGRAFÍA Y REFERENCIAS

6.1-Contenido para el desarrollo de la aplicación.

<https://fullstackopen.com/es/#course-contents>

<https://developer.mozilla.org/es/>

<https://laravel.com/docs/10.x>

<https://nodejs.org/es/docs>

<https://apuntes.de/nodejs-desarrollo-web/campos-virtuales-en-mongoose/#gsc.tab=0>

<https://www.github.com/>

6.2-Contenido para el despliegue de la aplicación.

<https://www.youtube.com/watch?v=DCoBcpOA7W4&t=172s>

<https://www.youtube.com/watch?v=1XIoUnAYCHY&t=547s>

<https://devopswithkubernetes.com/>

<https://kubernetes.io/es/docs/home/>

<https://hub.docker.com/>

<https://cloud.google.com/>

ANEXO 1: MANUAL DE INSTALACIÓN

1.1-Introducción.

Al contar el proyecto con tantos servidores independientes, es muy complejo de instalar localmente, para solucionar esto podemos instalar nuestro proyecto haciendo uso de los ficheros de Docker o Kubernetes proporcionados.

Podemos instalarlo localmente o en una máquina en la nube, para instalarlo localmente necesitamos tener Docker o Kubernetes instalado de manera local, para ello recomiendo la instalación de Docker Desktop, una aplicación de escritorio de Docker que te permite correr imágenes de Docker normales o bien usar la versión de Kubernetes que viene instalado con este y lanzar un despliegue usando un fichero de Kubernetes.

Puedes instalar Docker Desktop en el siguiente enlace: <https://www.docker.com/products/docker-desktop/>

Puedes encontrar los fichero del despliegue de Docker o Kubernetes mi repositorio de GitHub: https://github.com/GonzaloRando03/GonzaloShop_FullProject

1.2-Instalación del proyecto con Docker.

Para instalar el proyecto con Docker debemos dirigirnos a la carpeta de DockerDeployment del repositorio, aquí encontramos los ficheros necesarios para montar las imágenes y desplegar la aplicación en Docker.

```
> *** DockerDeploiment : bash — Konsole
Archivo Editar Ver Marcadores Complementos Preferencias Ayuda
gonzalo@DevPC:~/Trabajo/DockerProyecto/GonzaloShop_FullProject/DockerDeploiment$ ls -lc
total 64
-rw-r--r-- 1 root root 5731 feb 11 21:26 docker-compose.yml
drwxr-xr-x 2 root root 4096 feb 11 21:11 Frontend
drwxr-xr-x 2 root root 4096 feb 11 21:14 frontend-data
drwxr-xr-x 2 root root 4096 feb 11 21:11 GraphQL
drwxr-xr-x 2 root root 4096 feb 11 21:14 graphql-data
drwxr-xr-x 2 root root 4096 feb 11 21:11 JavaScript
drwxr-xr-x 2 root root 4096 feb 11 21:14 javascript-data
drwxr-xr-x 2 root root 4096 feb 11 21:11 mongo
drwxr-xr-x 2 root root 4096 feb 11 21:11 MongoDB
drwxr-xr-x 2 root root 4096 feb 11 21:11 MySQL
drwxr-xr-x 2 root root 4096 feb 11 21:11 PHP
drwxr-xr-x 2 root root 4096 feb 11 21:14 php-data
drwxr-xr-x 2 root root 4096 feb 11 21:12 Python
drwxr-xr-x 2 root root 4096 feb 11 21:14 python-data
drwxr-xr-x 7 999 999 4096 feb 11 21:27 schemas
gonzalo@DevPC:~/Trabajo/DockerProyecto/GonzaloShop_FullProject/DockerDeploiment$ 
```

El contenido de estos archivos es el explicado anteriormente por lo que no me voy a parar ahí. Para construir las imágenes de nuestros contenedores debemos ejecutar el comando “*docker-compose build*” dentro de este directorio.

```
> ... DockerDeploiment : docker-compose — Konsole
Archivo Editar Ver Marcadores Complementos Preferencias Ayuda
gonzalo@DevPC:~/Trabajo/DockerProyecto/GonzaloShop_FullProject/DockerDeploiment$ docker-compose build
Building frontend
[+] Building 1.1s (10/10) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 32B
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [internal] load metadata for docker.io/library/node:19-alpine3.16
=> [internal] load build context
=> => transferring context: 35B
=> [1/5] FROM docker.io/library/node:19-alpine3.16@sha256:d73a7d0f2ec5d9c0f4b8aeb7768151e0a3283edba46f
=> CACHED [2/5] RUN apk update
=> CACHED [3/5] RUN apk add nano git curl
=> CACHED [4/5] RUN git clone https://github.com/GonzaloRando03/GonzaloShop_Frontend.git
=> CACHED [5/5] COPY ./entrypoint.sh /
=> exporting to image
=> => exporting layers
=> => writing image sha256:48924ff829077ff286d266967afe9848d9339d84e7b0e102cb4f3c65bd894ba3
=> => naming to docker.io/library/dockerdeploiment_frontend
Building graphql
[+] Building 0.9s (10/10) FINISHED
=> [internal] load build definition from Dockerfile
```

Debemos esperar a que finalice el proceso.

```
=> => writing image sha256:042/c91ef93f14cb1431139b456/dafae4b8b/6b66/edda0d1b9/90025441ee8
=> => naming to docker.io/library/dockerdeploiment_mongodb
Building mysql
[+] Building 2.2s (6/6) FINISHED
=> [internal] load build definition from Dockerfile
=> => transferring dockerfile: 54B
=> [internal] load .dockerignore
=> => transferring context: 2B
=> [internal] load metadata for docker.io/library/mysql:8.0.13
=> [auth] library/mysql:pull token for registry-1.docker.io
=> CACHED [1/1] FROM docker.io/library/mysql:8.0.13@sha256:196c04e1944c5e4ea3ab86ae5f78f697cf
=> exporting to image
=> => exporting layers
=> => writing image sha256:25d6737cd7fa60dada06ad1f01b80f81257daa53c1c43402bc5ffd7c4f902917
=> => naming to docker.io/library/dockerdeploiment_mysql
gonzalo@DevPC:~/Trabajo/DockerProyecto/GonzaloShop_FullProject/DockerDeploiment$ █
```

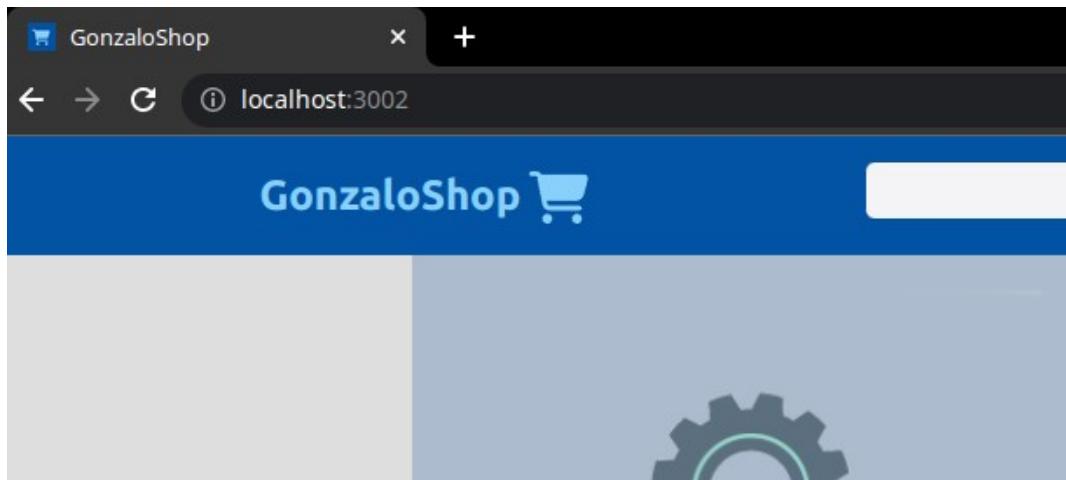
Si la construcción se ha realizado correctamente ya tendrás las imágenes de cada uno de los contenedores en tu Docker local, ahora solo tenemos que lanzar el despliegue de nuestro fichero *docker-compose.yml* con el comando “*docker-compose up*”.

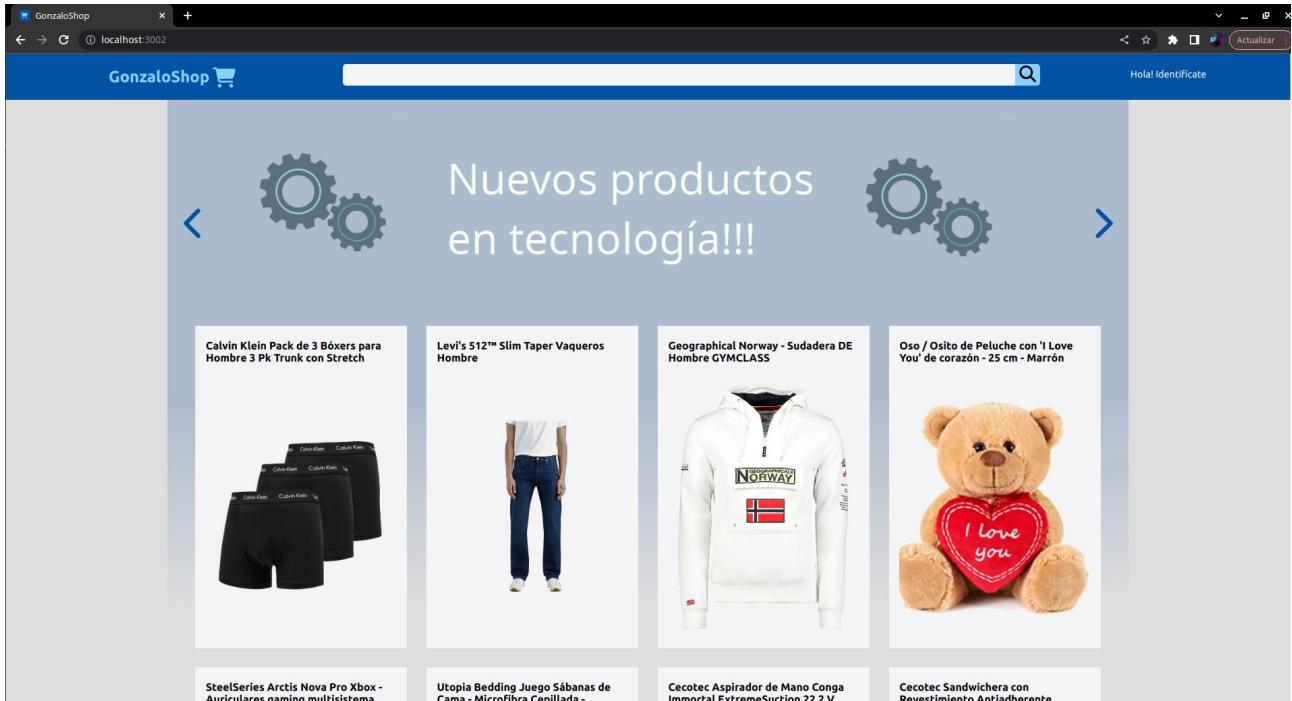
```
gonzalo@DevPC:~/Trabajo/DockerProyecto/GonzaloShop_FullProject/DockerDeployment$ docker-compose up
Starting user-microservice    ... done
Starting product-microservice ... done
Starting buy-microservice     ... done
Starting frontend             ... done
Starting mysql-d               ... done
Starting graphql-microservice ... done
Starting mongodb-d            ... done
Attaching to product-microservice, user-microservice, graphql-microservice, buy-microservice, mongodb
buy-microservice      | Loading composer repositories with package information
mongodb-d           | 2023-06-02T18:34:51.643+0000 I CONTROL [main] Automatically disabling TLS
mongodb-d           | 2023-06-02T18:34:51.649+0000 W ASIO      [main] No TransportLayer configure
mongodb-d           | 2023-06-02T18:34:51.649+0000 I CONTROL [initandlisten] MongoDB starting :
mongodb-d           | 2023-06-02T18:34:51.649+0000 I CONTROL [initandlisten] db version v4.2.9
mongodb-d           | 2023-06-02T18:34:51.649+0000 I CONTROL [initandlisten] allocator v2.2.1
mongodb-d           | 2023-06-02T18:34:51.649+0000 I CONTROL [initandlisten] options: { authSource: "admin", bindIp: "0.0.0.0", connectTimeoutMS: 30000, directConnection: false, fsync: true, host: "127.0.0.1", http://127.0.0.1:27017, logAppend: true, logLevel: "info", maxConns: 100, maxPoolSize: 100, maxWaitTimeMS: 15000, minPoolSize: 0, net: { port: 27017 }, oplog: { w: "majority" }, oplogSizeMB: 100, port: 27017, replicaset: null, security: { authMechanism: "SCRAM-SHA-256", keyFile: "/etc/mongod.key", mode: "disabled", roles: [ { db: "admin", permissions: [ { actions: [ "read", "write" ], resources: [ { db: "admin", collection: "*" } ] } ] } ], userFile: "/etc/mongod.users" }, processManagement: { pidFilePath: "/var/run/mongodb/mongod.pid" }, storage: { dbPath: "/var/lib/mongo", engine: "wiredTiger", journal: { enabled: true, type: "WiredTiger" } }, systemLog: { destination: "file", logAppend: true, path: "/var/log/mongodb/mongod.log" } }
```

Comienza el despliegue de la aplicación, el proceso finalizará en unos minutos y cuando lo haga, la aplicación estará corriendo correctamente.

<input type="checkbox"/>	<input checked="" type="checkbox"/>	 dockerdeploiment	-	Running (7/7)	More
<input type="checkbox"/>	<input type="checkbox"/>	 graphql-microservice 3a60167b4ae0	dockerdeploiment_gr	Running 4501:4000	29 seconds ago
<input type="checkbox"/>	<input type="checkbox"/>	 product-microservice 8401333cfb0a	dockerdeploiment_pr	Running 7501:7000	30 seconds ago
<input type="checkbox"/>	<input type="checkbox"/>	 user-microservice 8347be40dde1	dockerdeploiment_us	Running 3531:8000	29 seconds ago
<input type="checkbox"/>	<input type="checkbox"/>	 mongodb-d 6f799e674ac4	dockerdeploiment_mi	Running 12501:27017	30 seconds ago
<input type="checkbox"/>	<input type="checkbox"/>	 buy-microservice 24c95c70380c	dockerdeploiment_bu	Running 3551:3000	30 seconds ago
<input type="checkbox"/>	<input type="checkbox"/>	 frontend 4e5e8e9b785f	dockerdeploiment_frc	Running 3002:3000	29 seconds ago

Una vez desplegado podemos entrar en el navegador y acceder a la aplicación.



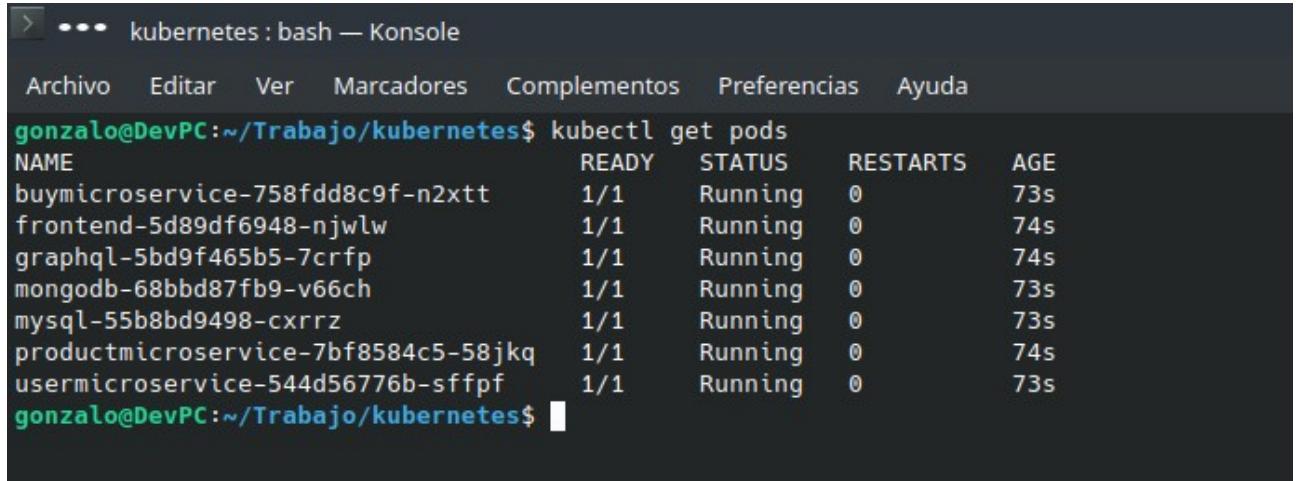


1.3-Instalación del proyecto con Kubernetes.

Para instalar y desplegar nuestro proyecto con Kubernetes debemos dirigirnos al directorio de KubernetesDeployment del repositorio, aquí encontramos el fichero con la configuración de nuestro despliegue de Kubernetes. Para lanzar la aplicación solo necesitamos correr dentro del directorio el comando “`kubectl apply -f <fichero>`”.

```
... kubernetes : bash — Konsole
Archivo Editar Ver Marcadores Complementos Preferencias Ayuda
gonzalo@DevPC:~/Trabajo/kubernetes$ kubectl apply -f gonzaloshop-deploy.yml
deployment.apps/frontend created
service/frontend unchanged
deployment.apps/graphql created
service/graphql unchanged
deployment.apps/productmicroservice created
service/productmicroservice unchanged
deployment.apps/buymicroservice created
service/buymicroservice unchanged
deployment.apps/usermicroservice created
service/usermicroservice unchanged
deployment.apps/mongodb created
persistentvolumeclaim/mongodb-pvc unchanged
deployment.apps/mysql created
persistentvolumeclaim/mysql-pvc unchanged
gonzalo@DevPC:~/Trabajo/kubernetes$
```

Solo con esto tendremos corriendo la aplicación. Si ejecutamos el comando “`kubectl get pods`” comprobaremos que todos los pods están funcionando correctamente.

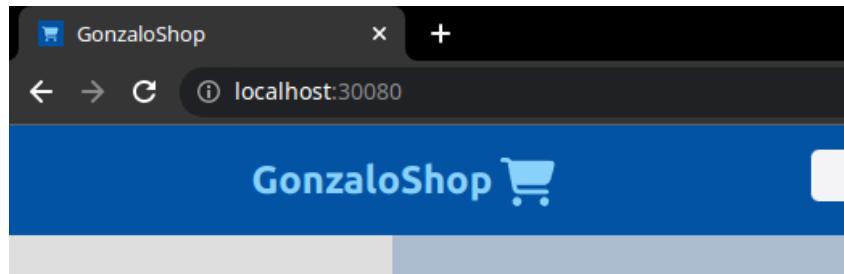


```

kubernetes : bash — Konsole
Archivo Editar Ver Marcadores Complementos Preferencias Ayuda
gonzalo@DevPC:~/Trabajo/kubernetes$ kubectl get pods
NAME                      READY   STATUS    RESTARTS   AGE
buymicroservice-758fdd8c9f-n2xtt   1/1     Running   0          73s
frontend-5d89df6948-njwlw        1/1     Running   0          74s
graphql-5bd9f465b5-7crfp       1/1     Running   0          74s
mongodb-68bbd87fb9-v66ch        1/1     Running   0          73s
mysql-55b8bd9498-cxrrz         1/1     Running   0          73s
productmicroservice-7bf8584c5-58jkq 1/1     Running   0          74s
usermicroservice-544d56776b-sffpf  1/1     Running   0          73s
gonzalo@DevPC:~/Trabajo/kubernetes$ 

```

Una vez desplegado podemos entrar en el navegador y acceder a la aplicación.



1.4-Instalación del proyecto en Google Cloud.

Vamos a desplegar nuestro proyecto en la nube usando Google Cloud, para ello necesitamos una cuenta en Google Cloud y crear un proyecto. Dentro de nuestro proyecto debemos ir a la gestión de Kubernetes y desde ahí creamos un nuevo cluster.

Nombre: gonzalo-shop-cluster
Región: europe-central2

Los nombres de los clústeres deben comenzar con una letra minúscula seguida por un máximo de 39 letras minúsculas, números o guiones. No puede terminar con un guion. No puedes cambiar el nombre del clúster una vez creado.

SIGUIENTE: REDES RESTABLECER CONFIGURACIÓN

Esperamos a que el cluster termine de iniciarse.

Clústeres de Kubernetes		+ CREAR	+ IMPLEMENTAR	C ACTUALIZAR	
DESCRIPCIÓN GENERAL		OBSERVABILIDAD	OPTIMIZACIÓN DE COSTOS		
Filtro Ingresar el nombre o el valor de la propiedad					
Estado	Nombre ↑	Ubicación	Modo	Cantidad de nodos	CPU virtuales totales
<input type="checkbox"/>	gonzalo-shop-cluster	europe-central2	Autopilot	0	0 GB

Debemos abrir la cloud shell y guardar el nuestro fichero de Kubernetes dentro de nuestra máquina de Google. Una vez hecho esto debemos arrancar nuestro despliegue al igual que en local.

```
gonzalorando03@cloudshell:~ (gonzaloshop)$ kubectl apply -f deploy-gonzaloshop.yml
Warning: Autopilot set default resource requests for Deployment default/frontend, as resource requests were not specified.
deployment.apps/frontend created
service/frontend created
Warning: Autopilot set default resource requests for Deployment default/graphql, as resource requests were not specified. See deployment.apps/graphql created
service/graphql created
Warning: Autopilot set default resource requests for Deployment default/productmicroservice, as resource requests were not specified. See deployment.apps/productmicroservice created
service/productmicroservice created
Warning: Autopilot set default resource requests for Deployment default/buymicroservice, as resource requests were not specified. See deployment.apps/buymicroservice created
service/buymicroservice created
Warning: Autopilot set default resource requests for Deployment default/usermicroservice, as resource requests were not specified. See deployment.apps/usermicroservice created
service/usermicroservice created
Warning: Autopilot set default resource requests for Deployment default/mysql, as resource requests were not specified. See deployment.apps/mysql created
persistentvolumeclaim/mysql-pvc created
service/mysql created
gonzalorando03@cloudshell:~ (gonzaloshop)$
```

Comprobamos que todos los pods estén corriendo correctamente.

```
gonzalorando03@cloudshell:~ (gonzaloshop)$ kubectl get pods
NAME                      READY   STATUS    RESTARTS   AGE
buymicroservice-785bfc948b-t88sh   1/1     Running   0          4m54s
frontend-9fc87d996-t7hc6        1/1     Running   0          4m55s
graphql-bf9ff596f-mwqwr         1/1     Running   0          4m55s
mysql-74684c9775-lr4xn        1/1     Running   0          4m54s
productmicroservice-8f977895d-p7j9s 1/1     Running   0          4m54s
usermicroservice-9bc99678d-zglfv   1/1     Running   0          4m54s
gonzalorando03@cloudshell:~ (gonzaloshop)$
```

Como vemos ahora nuestro cluster está consumiendo recursos ya que nuestra aplicación está corriendo.

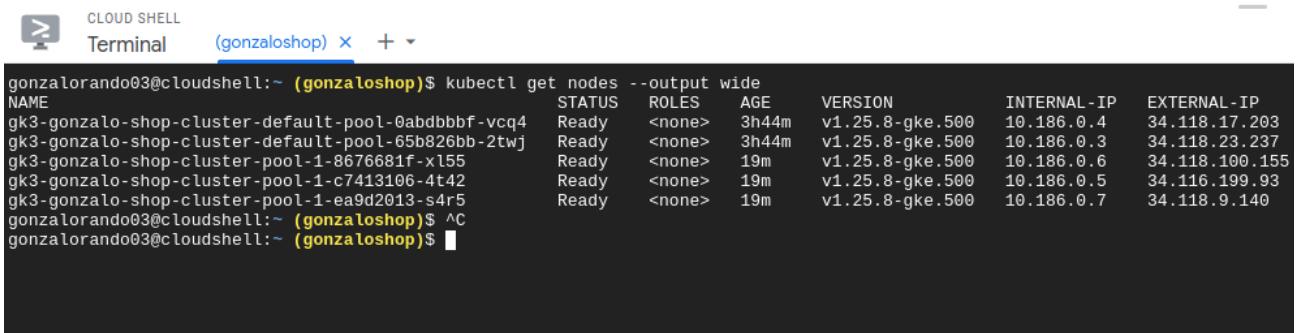
Estado	Nombre	Ubicación	Modo	Cantidad de nodos	CPU virtuales totales	Memoria total
<input type="checkbox"/>	<input checked="" type="checkbox"/> gonzalo-shop-cluster	europe-central2	Autopilot		3.33	12.66 GB

Nuestra aplicación está corriendo, pero todavía no podemos acceder a ella desde internet, para ello debemos abrir los puertos del Frontend y el servidor GraphQL mediante reglas de firewall.

```
gonzalorando03@cloudshell:~ (gonzaloshop)$ gcloud compute firewall-rules create test-node-port \
--allow tcp:30080
Creating firewall...working...Created [https://www.googleapis.com/compute/v1/projects/gonzaloshop/...
Creating firewall...done.
NAME: test-node-port
NETWORK: default
DIRECTION: INGRESS
PRIORITY: 1000
ALLOW: tcp:30080
DENY:
DISABLED: False
gonzalorando03@cloudshell:~ (gonzaloshop)$
```

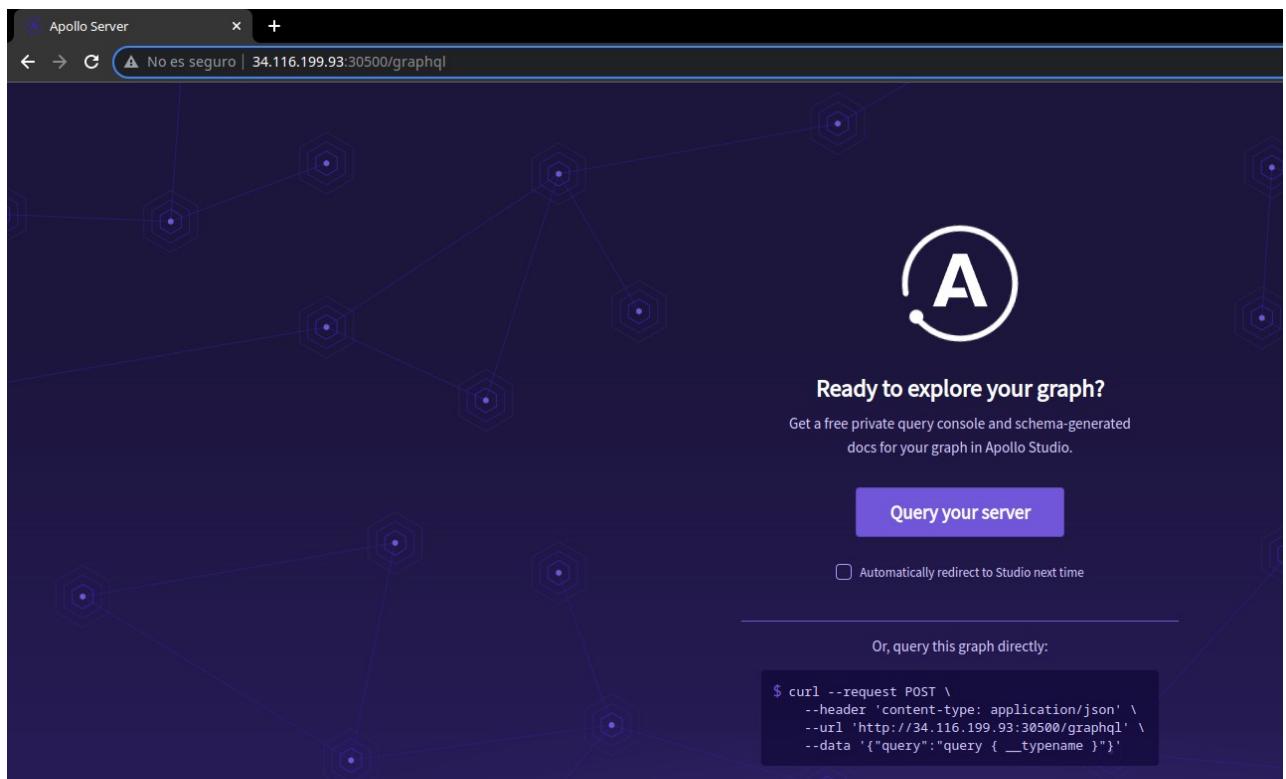
```
gonzalorando03@cloudshell:~ (gonzaloshop)$ gcloud compute firewall-rules create gql-node-port --allow tcp:30500
Creating firewall...working..Created [https://www.googleapis.com/compute/v1/projects/gonzaloshop/global/firewalls/gql-node-port]
Creating firewall...done.
NAME: gql-node-port
NETWORK: default
DIRECTION: INGRESS
PRIORITY: 1000
ALLOW: tcp:30500
DENY:
DISABLED: False
gonzalorando03@cloudshell:~ (gonzaloshop)$ █
```

Ya tenemos los puertos abiertos, por lo que si accedemos a la dirección IP pública de nuestro nodo podríamos acceder a nuestra aplicación. Para ver las direcciones de los nodos ejecutamos el siguiente comando: “`kubectl get nodes –output wide`”.

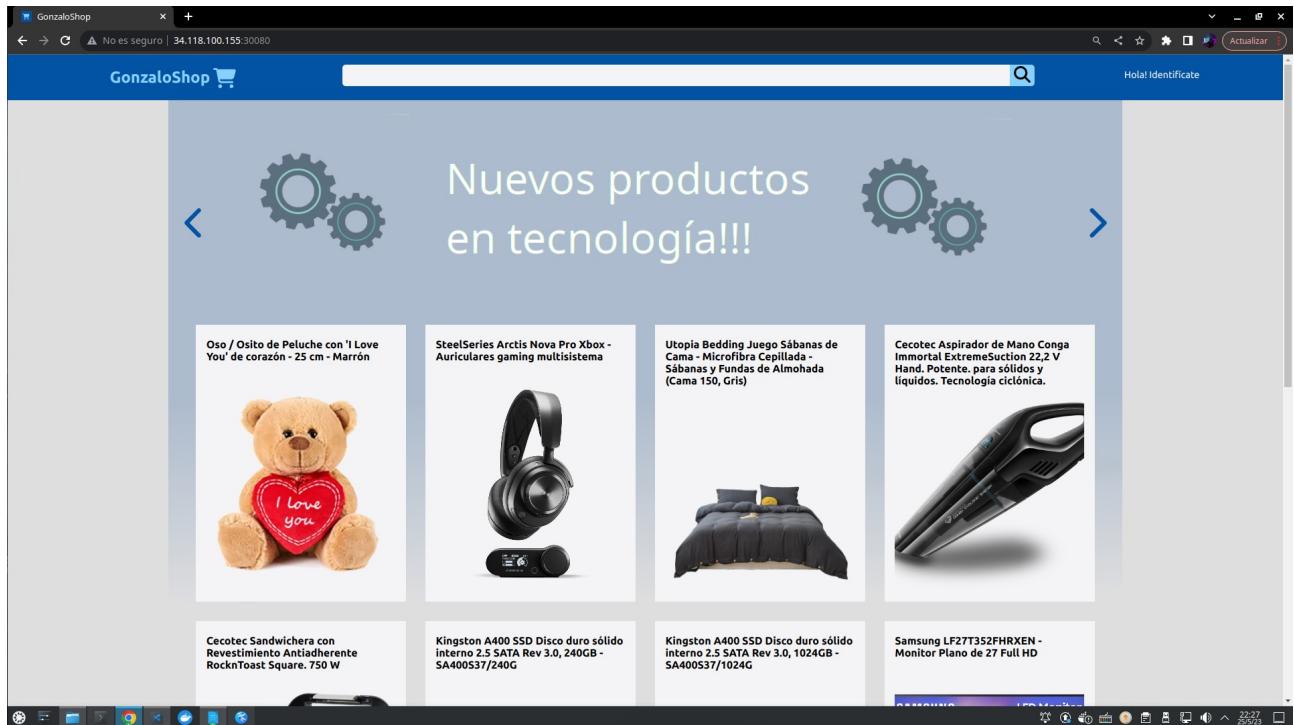


```
CLOUD SHELL
Terminal (gonzaloshop) X + ▾
gonzalorando03@cloudshell:~ (gonzaloshop)$ kubectl get nodes --output wide
NAME           STATUS   ROLES   AGE     VERSION   INTERNAL-IP   EXTERNAL-IP
gk3-gonzalo-shop-cluster-default-pool-0abdbbbf-vcq4   Ready    <none>   3h44m   v1.25.8-gke.500   10.186.0.4   34.118.17.203
gk3-gonzalo-shop-cluster-default-pool-65b826bb-2twj   Ready    <none>   3h44m   v1.25.8-gke.500   10.186.0.3   34.118.23.237
gk3-gonzalo-shop-cluster-pool-1-8676681f-xl55       Ready    <none>   19m    v1.25.8-gke.500   10.186.0.6   34.118.100.155
gk3-gonzalo-shop-cluster-pool-1-c7413106-4t42       Ready    <none>   19m    v1.25.8-gke.500   10.186.0.5   34.116.199.93
gk3-gonzalo-shop-cluster-pool-1-ea9d2013-s4r5       Ready    <none>   19m    v1.25.8-gke.500   10.186.0.7   34.118.9.140
gonzalorando03@cloudshell:~ (gonzaloshop)$ ^C
gonzalorando03@cloudshell:~ (gonzaloshop)$ █
```

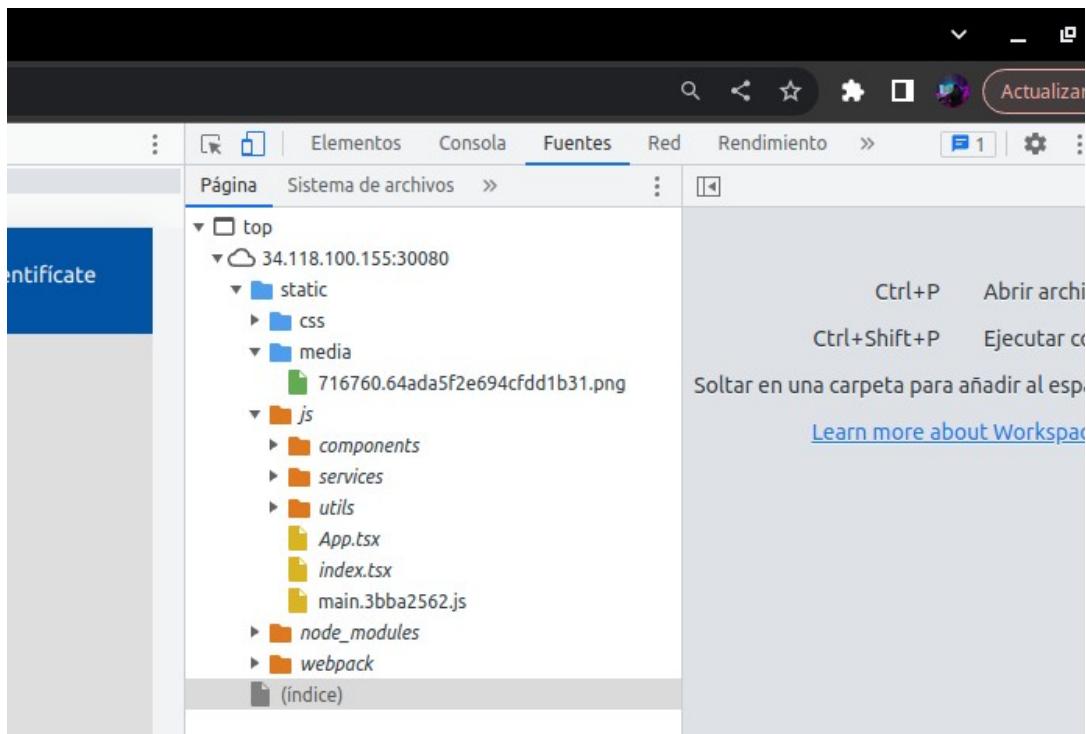
Intentemos acceder ahora a nuestro servidor GraphQL con su dirección IP pública.



Como vemos podemos acceder sin problema, por lo que si accedemos al Frontend tendíamos nuestra aplicación en internet totalmente funcional, desplegada en la nube de Google con Kubernetes y a la disposición de todo el mundo.



En esta captura demostramos que el servidor al que estamos accediendo es nuestro servidor de Google Cloud.



ANEXO 2: MANUAL DE USUARIO

2.1-Introducción.

GonzaloShop es una tienda online estilo Amazon o Aliexpress donde podrás navegar por gran cantidad de productos, realizar compras, publicar valoraciones, etc.

Cuenta con funciones como login y registro de usuarios, gestión de pagos mediante wallets, gestión de productos con buscador y filtros de búsqueda, sistema de valoraciones para los productos, carrito de la compra, entre otras funciones.

2.2-Registro y acceso.

En este apartado veremos como crear un usuario e iniciar sesión con este dentro de la aplicación.

2.2.1-Creación de una cuenta.

Pulsas el botón superior derecho de la cabecera donde dice “Hola! Identifícate”



En el menú pulsamos donde dice “¿No tienes cuenta? Regístrate.”



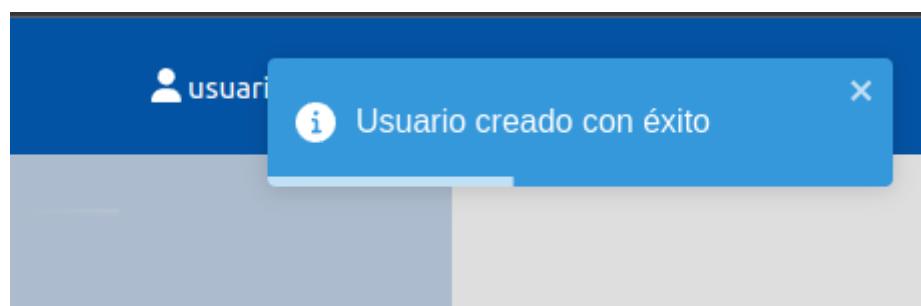
En el formulario debemos llenar todos los campos y pulsar en continuar.

The screenshot shows a registration form titled 'Crear cuenta' (Create account) on a website called 'GonzaloShop'. The form fields are as follows:

- Nombre**: Usuario
- Apellidos**: De prueba
- Correo electrónico**: usuariodeprueba@prueba.pru
- Número de cuenta**: 12345678901234567890
- Nombre de usuario**: usuarioprueba
- Contraseña**: (two password fields, both showing '.....')

Below the form is a note: "Todos los campos son obligatorios, pulsa continuar para registrarte." (All fields are mandatory, press continue to register). At the bottom are 'Continuar' (Continue) and 'Volver' (Back) buttons.

Debe aparecer un mensaje indicando que el usuario se ha creado correctamente.

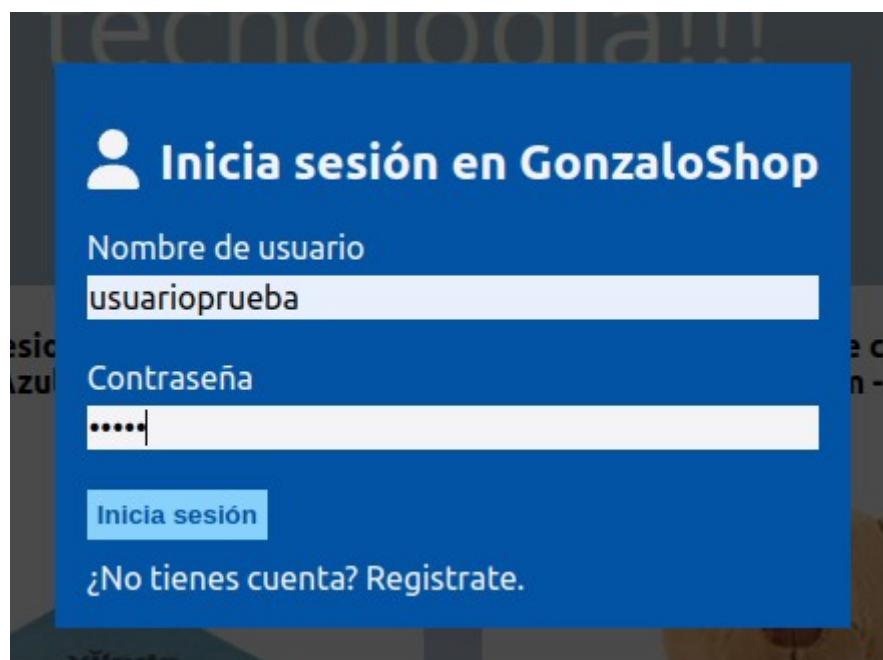


2.2.2-Iniciar sesión con una cuenta.

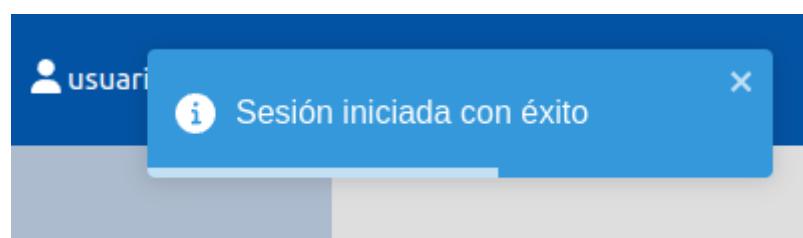
Pulsas el botón superior derecho de la cabecera donde dice “Hola! Identifícate”



Debemos introducir usuario y contraseña en el formulario y pulsar en inicia sesión.

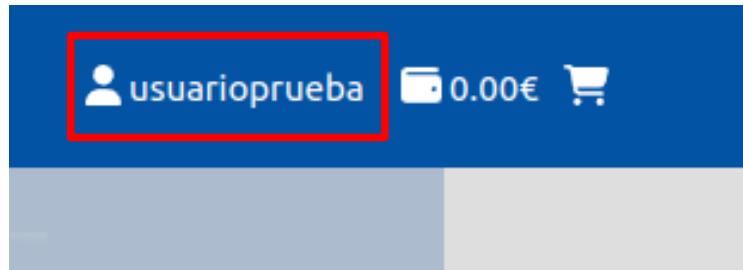


Debe aparecer un mensaje indicando que se ha iniciado sesión correctamente.

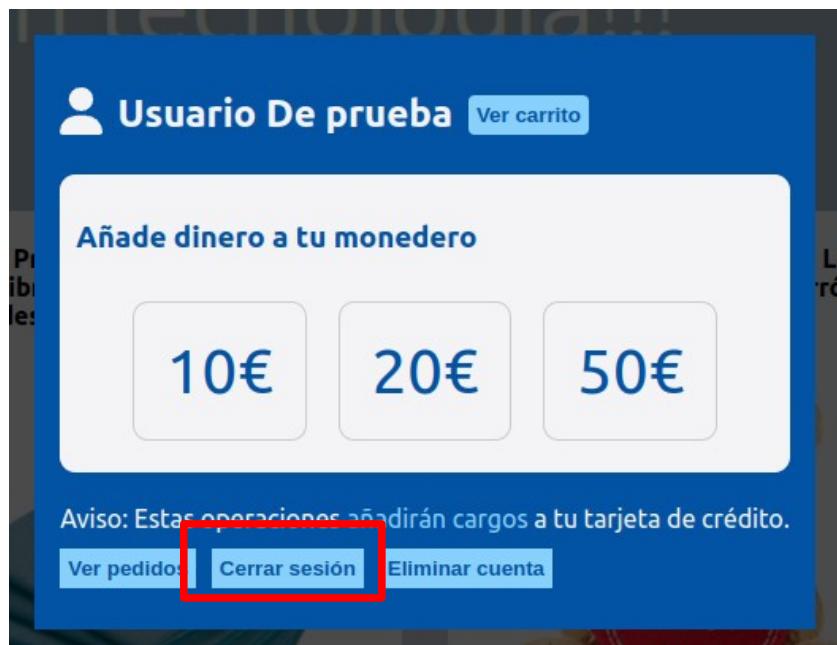


2.2.3-Cerrar sesión.

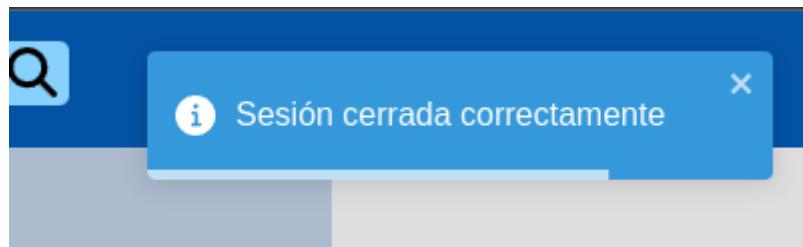
Pulsas el botón superior derecho de la cabecera donde aparece el nombre de tu usuario.



Debes pulsar el botón inferior central donde dice “Cerrar sesión”.



Debe aparecer un mensaje indicando que se ha cerrado sesión correctamente.

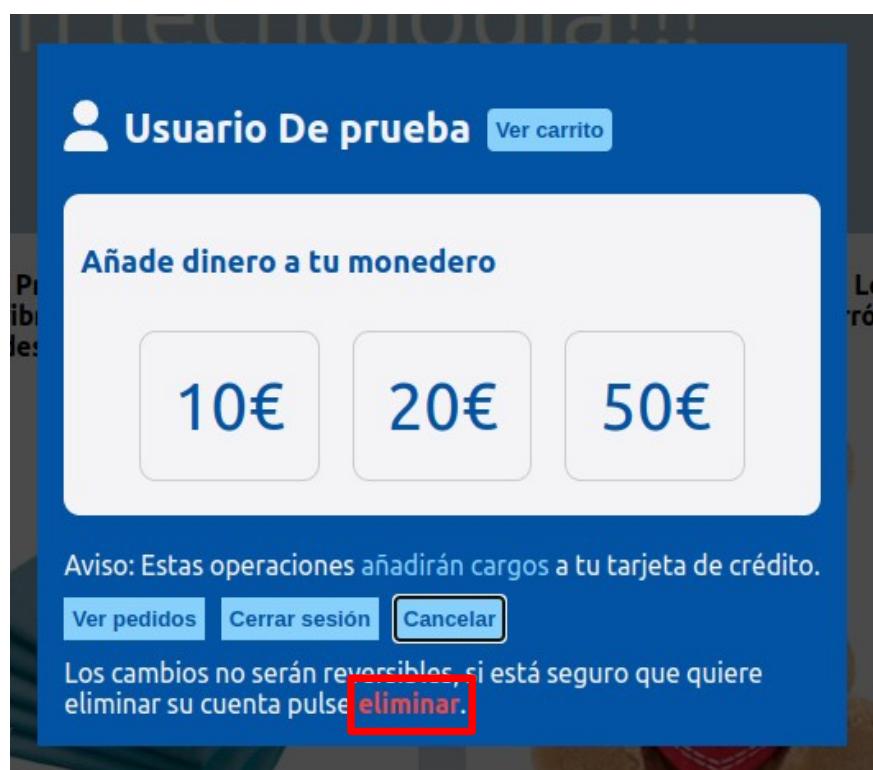


2.2.4-Borrar usuario.

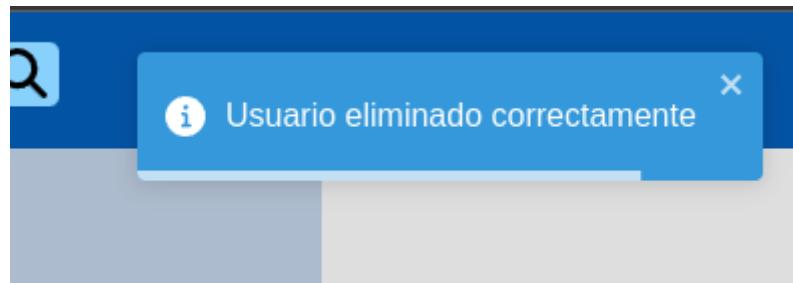
Para borrar usuario debemos pulsar el botón “Eliminar cuenta” del menú de usuario.



Para confirmar debemos pulsar el texto que dice “eliminar”.



Nos aparecerá un mensaje confirmando la correcta eliminación.



2.3-Explorar productos.

En este apartado veremos las distintas pantallas donde podemos explorar los productos disponibles en la tienda online y como utilizar los filtros de búsqueda.

2.3.1-Pantallas con productos.

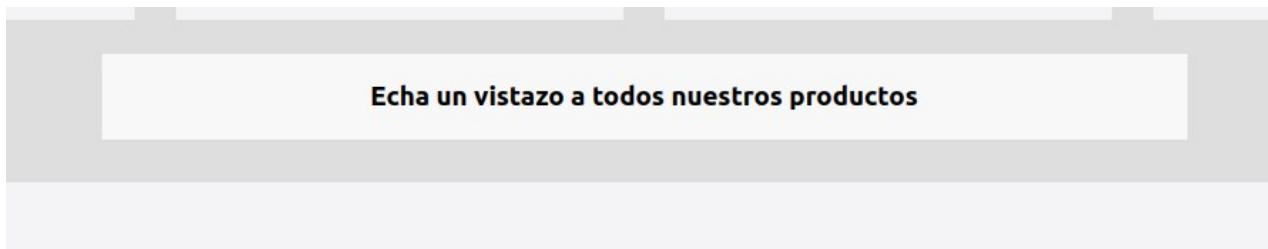
La primera pantalla visible con productos es el home de la tienda, que cuenta con una selección de productos destacados y otra de productos en oferta.



Para acceder a la pantalla principal de los productos existen dos formas, bien desde el buscador situado en la cabecera:



O bien desde el letrero con el texto “Echa un vistazo a todos nuestros productos” situado en el home:



Vista de la pantalla principal de productos:

Imagen	Nombre del Producto	Calificación	Precio
	Cecotec Sandwichera con Revestimiento Antiadherente RocknToast Square. 750 W	★★★★★	19.99€
	Vileda Professional 235385 Bayetas Microfibra Azul, 38 x 35cm, 5 Unidades	★★★★★	11.99€
	Oso / Osito de Peluche con 'I Love You' de corazón - 25 cm - Marrón	★★★★★	17.99€
	Kingston A400 SSD Disco duro sólido interno 2.5 SATA Rev 3.0, 480GB - SA400S37/480G	★★★★★	36.91€

2.3.2-Búsqueda de productos.

Podemos buscar productos desde cualquier pantalla escribiendo lo que deseemos en el buscador de la cabecera de la aplicación, desde un producto en concreto como “Disco duro kinston 500GB”, hasta algo más general como “Ropa”.

The screenshot shows the search results for 'ropa' (clothing) on the GonzaloShop website. The search bar at the top contains the word 'ropa'. On the left, there is a sidebar with filtering options: 'Filtros' (Filters), 'Categoría' (Category), 'Ordenar por' (Sort by), 'Precio' (Price), and 'Ofertas' (Offers). The main area displays three products:

- Geographical Norway - Sudadera DE Hombre GYMCLASS**: A white hoodie with a 'NORWAY' logo and a small flag. Rating: ★★★★☆. Price: 34.89€.
- adidas Core18 Hoody Sudadera Hombre**: A black hoodie with a small logo on the chest. Rating: ★★★★☆. Price: 26.95€.
- Calvin Klein Pack de 3 Bóxers para Hombre 3 Pk Trunk con Stretch**: Three black Calvin Klein boxer shorts. Rating: ★★★★☆. Price: 25€.

Podemos personalizar nuestras búsquedas usando los filtros situados en el panel lateral izquierdo.

2.3.3-Filtros y opciones de ordenamiento.

Desde este panel podemos filtrar nuestras búsquedas siguiendo distintos criterios como categoría, precio o productos en oferta.

Ejemplo de filtro usando estos tres criterios (producto en oferta de la categoría Tecnología con un precio entre 100 y 130€) :

The screenshot shows the search results for 'Tecnología' (Technology) on the GonzaloShop website. The search bar at the top contains the word 'Tecnología'. On the left, there is a sidebar with filtering options: 'Filtros' (Filters), 'Categoría' (Category), 'Ordenar por' (Sort by), 'Precio' (Price), and 'Ofertas' (Offers). The main area displays one product, which is highlighted in a larger box:

- Xiaomi Redmi 9C Midnight Gray 3GB RAM 64GB ROM**: A smartphone shown from both the back and front. Rating: ★★★★☆. Price: 125€.

También podemos ordenar nuestros resultados por criterios como: productos destacados, precio más alto o bajo y por valoración.

Ejemplo de ordenación por precio más bajo:

The screenshot shows the GonzaloShop application interface. On the left, there is a sidebar with filtering options: 'Filtros' (Filters), 'Categoría' (Category), 'Ordenar por' (Sort by), 'Precio' (Price), and 'Ofertas' (Offers). The 'Ordenar por' section is set to 'Precio más bajo' (Lowest price). The main area displays a grid of four products:

- Abanderado Termal Algodón Invierno C/Redondo Camiseta térmica para Hombre**
★ ★ ★ ★ ★
9.99€
- Vileda Profesional 235385 Bayetas Microfibra Azul, 38 x 35cm, 5 Unidades**
★ ★ ★ ★ ★
11.99€
- adidas 3-Stripes tee T-Shirt Hombre**
★ ★ ★ ★ ★
14.99€
- Peluche de Kaito**
★ ★ ★ ★ ★
15.95€

2.4-Detalle del producto.

En este apartado veremos la pantalla donde aparece un producto individual, para acceder a ella solo debemos pulsar en cualquier producto que aparece en la aplicación.

2.4.1-Información del producto.

Al acceder a un producto nos aparece la siguiente pantalla con información muy relevante como el nombre, la marca, el precio, las características, valoraciones y imágenes.

Kingston A400 SSD Disco duro sólido interno 2.5 SATA Rev 3.0, 240GB - SA400S37/240G

★★★★★ 2 valoraciones

23.91€

Descripción: Diversas capacidades, con suficiente espacio para aplicaciones o para sustituir un disco duro

Marca: Kingston

Colores: negro

Conectores: SATA3

Capacidad: 240GB

Valoraciones

paco_gonzalez ★★★★★ Muy buen producto!

pruba7 ★★★★★ Muy buen producto!

La información importante del producto la encontramos en el panel lateral derecho y leyéndola podemos hacer una idea bastante precisa sobre el producto que estamos consultando.

Las imágenes se encuentran en el panel lateral izquierdo y si pulsamos en cualquiera de ellas, podemos verla mas detalladamente.

Kingston A400 SSD Disco duro sólido interno 2.5 SATA Rev 3.0, 240GB - SA400S37/240G

★★★★★ 2 valoraciones

23.91€

Descripción: Diversas capacidades, con suficiente espacio para aplicaciones o para sustituir un disco duro

Marca: Kingston

Colores: negro

Conectores: SATA3

Capacidad: 240GB

Valoraciones

paco_gonzalez ★★★★★ Muy buen producto!

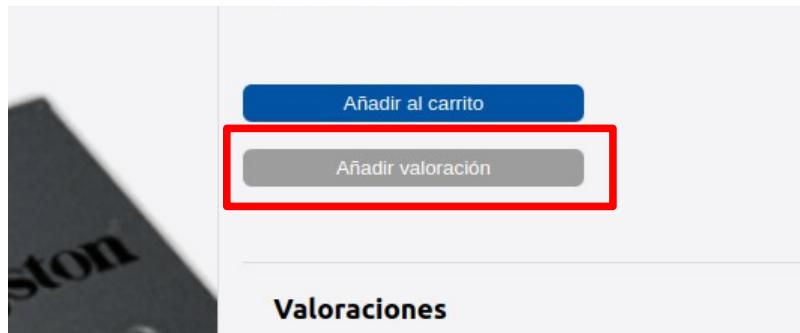
pruba7 ★★★★★ Muy buen producto!

2.4.2-Valoración del producto.

Debajo de las características encontramos las valoraciones de los usuarios donde podremos ver lo que opinan del producto los compradores, cuenta con un sistema de estrellas donde cada usuario valora con un número del 1 al 5 y se calcula una media de estas que aparece debajo del nombre del producto.

The screenshot shows a product listing for a Kingston A400 SSD. At the top, it displays the product name: "Kingston A400 SSD Disco duro sólido interno 2.5 SATA Rev 3.0, 240GB - SA400S37/240G". Below the name is a rating section showing a yellow star icon followed by "2 valoraciones". Underneath the rating is the price "23.91€". In the middle of the page, there is a "Valoraciones" (Reviews) section. It contains two reviews: one from "paco_gonzalez" with a 5-star rating and the comment "Muy buen producto!", and another from "pruba7" with a 5-star rating and the comment "Muy buen disco".

Para añadir una valoración debemos pulsar el botón de “Añadir valoración”.



En el formulario que nos aparece debemos escribir nuestra valoración e indicar las estrellas.

The screenshot shows a "Añade tu valoración" (Add your review) form. It has a text area labeled "Escribe lo que opinas" (Write what you think) containing the text "valoración de prueba". Below that is a dropdown menu labeled "Selecciona las estrellas" (Select stars) set to "5". At the bottom is a blue "Enviar valoración" (Send review) button, which is highlighted with a red rectangular box.

Pulsa en enviar valoración y la valoración se publicará automáticamente.



2.5-Funciones del carrito y compra del producto.

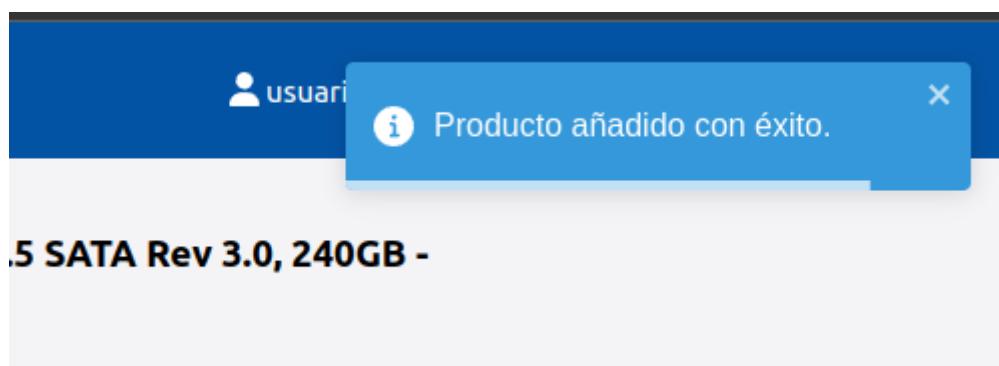
En este apartado veremos como funciona el carrito de la compra y como podemos comprar productos en GonzaloShop.

2.5.1-Agregar productos al carrito.

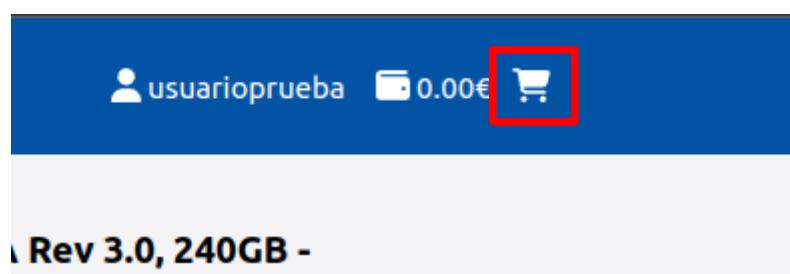
Para agregar un producto al carrito simplemente debes pulsar el botón que se encuentra encima del botón de “Añadir valoración”.



Nos aparece un mensaje indicando que el producto se ha guardado en el carrito correctamente.



Para ver nuestro carrito podemos pulsar el botón con forma de carro de la compra que se encuentra en la esquina superior derecha de la cabecera.



O bien en el botón de “Ver carrito” del menú del usuario.



Dentro del carrito encontramos nuestros datos de compra y la lista de productos que tenemos guardados.

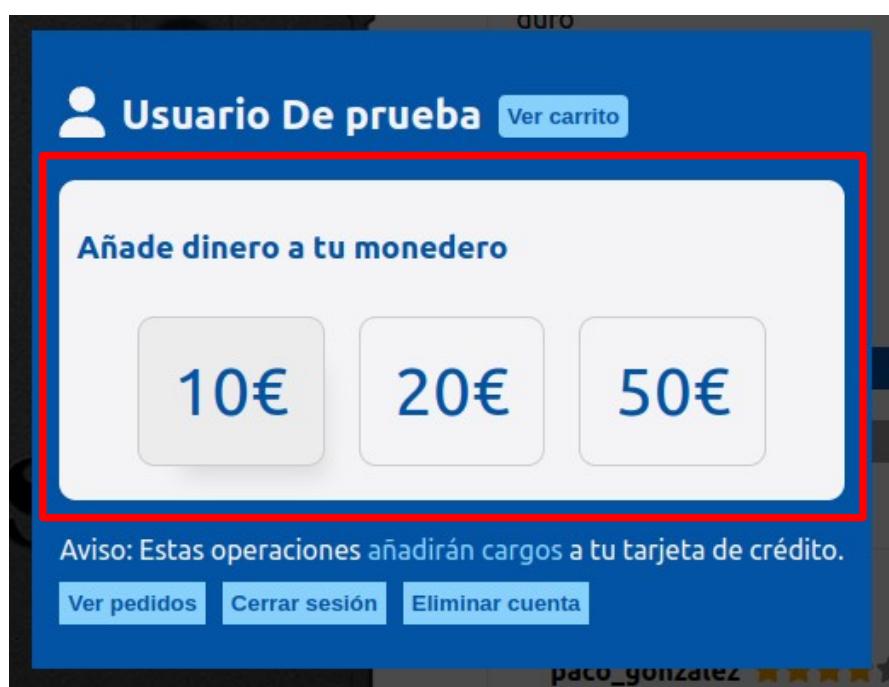
A screenshot of a web-based shopping cart page. On the left, there's a sidebar with "Productos" and a list item for "Kingston A400 SSD Disco duro sólido interno 2.5 SATA Rev 3.0, 240GB - SA400S37/240G x1" with a small image of the SSD. Below it, the price is listed as "Precio: 23.91€" and there's a "Eliminar producto" button. At the bottom, it says "Precio total: 23.91€". On the right, the main content area is titled "Carrito de la compra de usuarioprueba.". It shows the product details again, along with a discount message: "Descuento: 20% de nuevo usuario debe aplicarse" and "Precio final: 19.13€". There's also a "Número de cuenta: 12345*****" field, a "Dirección de envío:" label with a text input field, and two buttons: "Seguir comprando" and "Hacer pedido".

Podemos eliminar productos pulsando el botón de eliminar debajo de la imagen de cada producto.

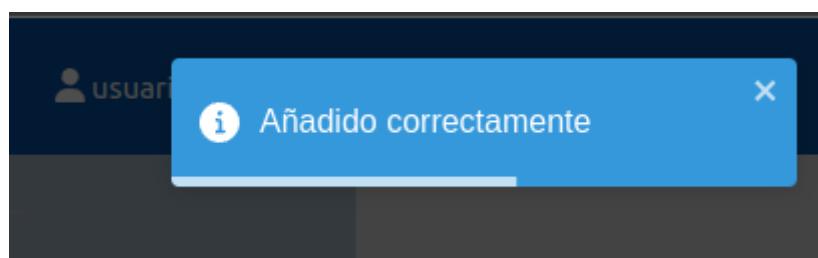


2.5.2-Proceso de compra.

Para comprar un producto necesitas añadir dinero a tu monedero, para ello debes comprar una tarjeta dentro del panel de usuario.



Puedes comprar tarjetas de 10€, 20€ y 50€, al comprar cualquiera de ellas se te añadirá el dinero al monedero y aparecerá un mensaje.





Si volvemos al carrito podemos realizar una compra.

GonzaloShop 🛒

Carrito de la compra de usuarioprueba.

Productos: 1

Precio del carrito: 23.91€

Descuento: 20% de nuevo usuario debe aplicarse

Precio Final: 19.13€

Número de cuenta: 12345*****

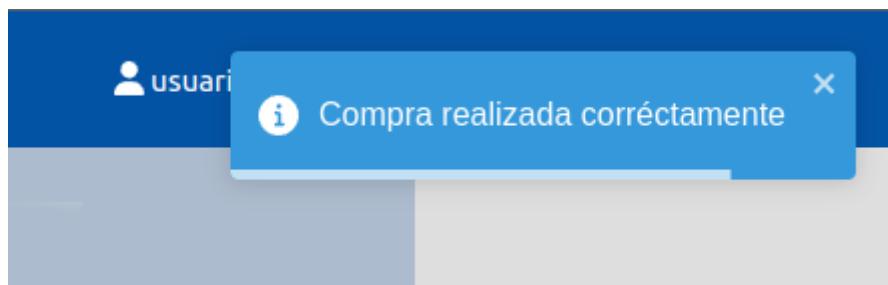
Dirección de envío:

Cº/dirección de prueba Nº1

Seguir comprando

Hacer pedido

Debemos indicar la dirección a la que queremos enviar nuestro producto y realizamos el pedido. Nos debe aparecer un mensaje indicando que la compra se ha realizado correctamente.



2.5.3-Gestión de pedidos.

Para ver los pedidos debemos pulsar el botón de “Ver pedidos” situado en el menú del usuario.



En la pantalla tenemos una tabla con todos los pedidos del usuario.

Pedidos de usuarioprueba:					
Nº	Fecha de la compra	Fecha de entrega	Precio Total	Estado del pedido	Artículos
1	2023-06-05	2023-06-20	19.13€	Pedido en proceso	Nombre: Kingston A400 SSD Disco duro sólido interno 2.5 SATA Rev 3.0, 240GB - SA400S37/240G Cantidad: 1 Precio: 23.91€

En esta tabla podemos ver información como la fecha de compra, la fecha prevista de entrega, el precio total de la compra, el estado del pedido, y la información sobre los artículos. Desde esta pantalla podemos consultar el estado de los pedidos y el tiempo que falta para su entrega.