

# Taller 2 - Pruebas y lanzamiento

## - Reporte de resultados -

---

dev 4 Branches 0 Tags

Go to file Add file Code

This branch is 10 commits ahead of SelimHorri/ecommerce-microservice-backend-app:master . Contribute Sync fork

NightParker725 fix in cicd pipes ✓ d796652 · 34 minutes ago 462 Commits

.github/workflows	fix in cicd pipes	34 minutes ago
.mvn/wrapper	init	4 years ago
api-gateway	added remaining tests in user, order and other	2 hours ago
cloud-config	remove serialization indent	4 years ago
favourite-service	added remaining tests in user, order and other	2 hours ago
k8s	added CICD pipelines and k8 config for branching new	52 minutes ago
order-service	added remaining tests in user, order and other	2 hours ago
payment-service	added remaining tests in user, order and other	2 hours ago
product-service	added remaining tests in user, order and other	2 hours ago
proxy-client	added remaining tests in user, order and other	2 hours ago
service-discovery	added remaining tests in user, order and other	2 hours ago

## Contexto

De forma resumida, este taller consiste en un sistema **e-commerce basado en microservicios** (fuente original:

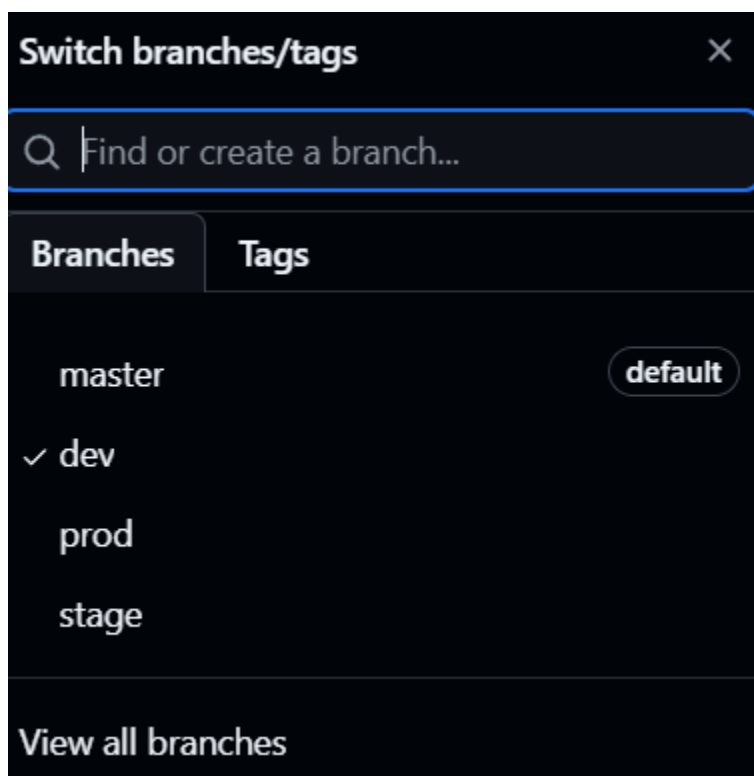
<https://github.com/SelimHorri/ecommerce-microservice-backend-app>), desarrollado con **Spring Boot y arquitectura en la nube**, y pues cada servicio cumple una función específica dentro de la arquitectura del servidor completo (usuarios, productos, pedidos, autenticación, etc.).

---

---

## Configuración del taller:

En primer lugar, se estableció una estrategia de branching, dividido entre dev, prod/master y stage, siendo estas para desarrollo local, producción y preproducción respectivamente, Así mismo, sin contar los servicios básicos de api-gateway, proxy y cloud, se enfocó el desarrollo, tests y rendimiento en los microservicios de products, users y orders, aunque de todos modos todos los microservicios fueron desplegados (localmente) y funcionales (también localmente, solo fueron llevados a mini kube los mencionados de enfoque).



En segundo lugar, se establecieron los archivos de despliegue local para docker y jenkins, y junto a estos los yml para pipelines de desarrollo y arquitectura en github actions y que corrieran también en jenkins.

## Parte del Docker compose usado:

```
compose.yml
You, 3 days ago | 2 authors (You and one other)
1 version: '3'
  Run All Services
2 services:
  Run Service
3 api-gateway-container:
4   build: ./api-gateway
5   ports:
6     - 8080:8080
7   networks:
8     - microservices_network
9   environment:
10    - SPRING_PROFILES_ACTIVE=dev
11    - SPRING_ZIPKIN_BASE-URL=http://zipkin:9411
12    - SPRING_CONFIG_IMPORT=optional:configserver:http://cloud-config-container:9296/
13    - EUREKA_CLIENT_REGION=default
14    - EUREKA_CLIENT_AVAILABILITYZONES_DEFAULT=myzone
15    - EUREKA_CLIENT_SERVICEURL_MYZONE=http://service-discovery-container:8761/eureka
16    - EUREKA_CLIENT_SERVICEURL_DEFAULTZONE=http://service-discovery-container:8761/eureka/
17
  Run Service
18 user-service-container:
19   build: ./user-service
20   ports:
21     - 8700:8700
22   networks:
23     - microservices_network
24   environment:
25    - SPRING_PROFILES_ACTIVE=dev
26    - SPRING_ZIPKIN_BASE-URL=http://zipkin:9411
27    - SPRING_CONFIG_IMPORT=optional:configserver:http://cloud-config-container:9296/
28    - EUREKA_CLIENT_REGION=default
29    - EUREKA_CLIENT_AVAILABILITYZONES_DEFAULT=myzone
30    - EUREKA_CLIENT_SERVICEURL_MYZONE=http://service-discovery-container:8761/eureka
31    - EUREKA_CLIENT_SERVICEURL_DEFAULTZONE=http://service-discovery-container:8761/eureka/
```

Ejemplo de parte de un jenkinsfile (pipeline de build & test para product en este caso:

```

product-service > Jenkinsfile
...
1  pipeline {
2    agent any
3
4    environment {
5      PROJECT_NAME = "product-service"
6      JAR_NAME = "product-service-v0.1.0.jar"
7    }
8
9    stages {
10   stage('Checkout') {
11     steps {
12       checkout scm
13     }
14   }
15
16   stage('Build with Maven') {
17     steps {
18       dir("${PROJECT_NAME}") {
19         sh './mvnw clean package'
20       }
21     }
22   }
23
24   stage('Run Unit Tests') {
25     steps {
26       dir("${PROJECT_NAME}") {
27         sh './mvnw test'
28       }
29     }
30   }
31
32   stage('Build Docker Image') {
33     steps {

```

En workflows para dev luce para el build & test de todos los micro servicios asi (ejecutandose en dev) :

```

.github > workflows > ci-dev.yml
...
1  name: CI - Dev
2
3  on:
4  push:
5    branches: [ "dev" ]
6
7  jobs:
8  build-test:
9    runs-on: ubuntu-latest
10   steps:
11     - uses: actions/checkout@v3
12     - name: Set up JDK 17
13       uses: actions/setup-java@v3
14       with:
15         distribution: 'temurin'
16         java-version: 17
17     - name: Build & Test
18       run: mvn -B clean verify
19

```

En tercer lugar y como ya mencioné, los servicios usaron docker, y es que se ejecutan en contenedores, orquestados con **Kubernetes**.

---

Cada microservicio tiene su propio **dockerfile ya establecido**, y los manifiestos de Kubernetes están en la **k8s a partir de minikube**.

### Ejemplo de parte de un dockerfile:











```
product-service > Dockerfile > ...
You, 3 days ago | 2 authors (You and one other)
1 FROM openjdk:11
2 WORKDIR /home/app
3 ENV SPRING_PROFILES_ACTIVE=dev
4 COPY target/product-service-v0.1.0.jar app.jar
5 EXPOSE 8500
6 ENTRYPOINT ["java", "-Dspring.profiles.active=${SPRING_PROFILES_ACTIVE}", "-jar", "app.jar"]
7 |
```

### Ejemplo del pipeline de construcción de las imágenes de docker (en la rama stage - preproducción)

```
.github > workflows > cd-stage.yml
2
3 on:
4   push:
5     branches: [ "stage" ]
6
7 jobs:
8   build-push:
9     runs-on: ubuntu-latest
10    steps:
11      - uses: actions/checkout@v3
12      - name: Log in to DockerHub
13        uses: docker/login-action@v2
14        with:
15          username: ${ secrets.DOCKERHUB_USERNAME }
16          password: ${ secrets.DOCKERHUB_TOKEN }
17      - name: Build and push Docker images
18        run: |
19          for dir in user-service product-service order-service shipping-service favourite-service payment-service; do
20            docker build -t ${ secrets.DOCKERHUB_USERNAME }/$dir:stage ./dir
21            docker push ${ secrets.DOCKERHUB_USERNAME }/$dir:stage
22          done
23
```

En este caso, se utilizan los **secrets** definidos en el repositorio remoto que contienen mis credenciales y token de docker, y como se ve, construye todos los servicios en su totalidad (a pesar de que como dije, el enfoque en pruebas y rendimiento solo recae en tres de esos).

---

Repository secrets		New repository secret	
Name 		Last updated	
 DOCKERHUB_TOKEN		4 hours ago	 
 DOCKERHUB_USERNAME		4 hours ago	 
 KUBECONFIG_CONTENTS		4 hours ago	 

Para los kubernetes, aquí hay un ejemplo para user-service:

```
k8s > ! deployment-user.yaml
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: user-service
5    namespace: ecommerce
6  spec:
7    replicas: 1
8    selector:
9      matchLabels:
10       app: user-service
11   template:
12     metadata:
13       labels:
14         app: user-service
15     spec:
16       containers:
17       - name: user-service
18         image: docker.io/nightparker725/user-service:prod
19       ports:
```

Aquí para la api-gateway:

```
k8s > ! api-gateway.yaml
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: api-gateway
5  spec:
6    replicas: 1
7    selector:
8      matchLabels:
9        app: api-gateway
10   template:
11     metadata:
12       labels:
13         app: api-gateway
14     spec:
15       containers:
16         - name: api-gateway
17           image: nightparker725/api-gateway:latest
18           ports:
19             - containerPort: 8080
20 ---
21 apiVersion: v1
22 kind: Service
23 metadata:
24   name: api-gateway
25 spec:
26   selector:
27     app: api-gateway
28   ports:
29     - port: 8080
30       targetPort: 8080
31       nodePort: 30080
32   type: NodePort
```

En estos básicamente se accede a la imagen montada de docker del servicio para ser configurada y usada en kubernetes con minikube, toda esta configuración en la carpeta de k8s.

## Resultados:

Por último, podemos confirmar que todo salió bien a partir de varios comandos, por ejemplo consultando las imágenes construidas:

### Listado de imágenes construidas para el proyecto y su despliegue

```
C:\Users\reyda\Desktop\ecommerce-microservice-backend-app>docker images
```

REPOSITORY	TAG	IMAGE ID	CREATED	SIZE
ecommerce-microservice-backend-app-shipping-service-container	latest	8574a11a91c5	2 days ago	1.18GB
ecommerce-microservice-backend-app-user-service-container	latest	2eb252c5b692	2 days ago	1.18GB
ecommerce-microservice-backend-app-payment-service-container	latest	498392f3843f	2 days ago	1.18GB
ecommerce-microservice-backend-app-order-service-container	latest	9badffa60c28	2 days ago	1.18GB
ecommerce-microservice-backend-app-api-gateway-container	latest	f57734fbcfc3	2 days ago	1.14GB
ecommerce-microservice-backend-app-product-service-container	latest	18d1bf5b9a11	2 days ago	1.18GB
product-service	latest	2528c72546cc	2 days ago	1.18GB
user-service	latest	c08829940fc9	3 days ago	1.18GB
jenkins/jenkins	latest	3c2f4a0a573a	8 days ago	810MB

### Servicios y pods por kubernetes desplegados y corriendo (para stage)

```
C:\Users\reyda\Desktop\ecommerce-microservice-backend-app>kubectl get services
```


NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
api-gateway	NodePort	10.105.118.115	<none>	8080:30080/TCP	99s
kubernetes	ClusterIP	10.96.0.1	<none>	443/TCP	174m
product-service	NodePort	10.97.49.3	<none>	8082:30082/TCP	99s
















































```
C:\Users\reyda\Desktop\ecommerce-microservice-backend-app>kubectl get services
```

NAME	TYPE	CLUSTER-IP	EXTERNAL-IP	PORT(S)	AGE
api-gateway	NodePort	10.105.118.115	<none>	8080:30080/TCP	99s
kubernetes	ClusterIP	10.96.0.1	<none>	443/TCP	174m
product-service	NodePort	10.97.49.3	<none>	8082:30082/TCP	99s
user-service	NodePort	10.110.154.243	<none>	8081:30081/TCP	98s

## Despliegue de los servicios exitoso por docker desktop (local)

Container CPU usage 15.94% / 1200% (12 CPUs available) Container memory usage 10.76GB / 15.15GB Show ch

Search  Only show running containers

<input type="checkbox"/>	Name	Container ID	Image	Port(s)	CPU (%)	Last start...	Actions
<input type="checkbox"/>	 jenkins	ac7b119720fe	<a href="#">jenkins/jenkins:lts-jdk</a>	50000:50000  <a href="#">Show all ports (2)</a>	3.15%	27 seconds ago	 
<input type="checkbox"/>	 minikube	f065eb9da1a2	<a href="#">k8s-minikube/kicbas</a>	61147:22  <a href="#">Show all ports (5)</a>	10.92%	3 hours ago	 
<input type="checkbox"/>	 ecommerce-microservice-backend-app	-	-	-	1.91%	5 hours ago	 
<input type="checkbox"/>	 order-service-container-1	c0b0b02dc885	<a href="#">ecommerce-microse</a>	8300:8300 	0.12%	5 hours ago	 
<input type="checkbox"/>	 product-service-container-1	77de6ff53c46	<a href="#">ecommerce-microse</a>	8500:8500 	0.12%	5 hours ago	 
<input type="checkbox"/>	 user-service-container-1	24054c08605f	<a href="#">ecommerce-microse</a>	8700:8700 	0.11%	5 hours ago	 
<input type="checkbox"/>	 payment-service-container-1	2d60127f52bc	<a href="#">ecommerce-microse</a>	8400:8400 	0.14%	5 hours ago	 
<input type="checkbox"/>	 api-gateway-container-1	059a9c4fedf8	<a href="#">ecommerce-microse</a>	8080:8080 	0.11%	5 hours ago	 
<input type="checkbox"/>	 zipkin-1	b2258a3b6717	<a href="#">openzipkin/zipkin</a>	9411:9411 	0.12%	5 hours ago	 
<input type="checkbox"/>	 shipping-service-container-1	c8593c35ed41	<a href="#">ecommerce-microse</a>	8600:8600 	0.14%	5 hours ago	 
<input type="checkbox"/>	 service-discovery-container-1	8a14f0dbdec3	<a href="#">selimhorri/service-di</a>	8761:8761 	0.9%	5 hours ago	 
<input type="checkbox"/>	 cloud-config-container-1	422bc77b7aa5	<a href="#">selimhorri/cloud-con</a>	9296:9296 	0.15%	5 hours ago	 



Builds [Give feedback](#)

Selected builder  
desktop-linux

Import builds

Builder settings

Build large and multi-platform Docker images faster in Docker Build Cloud.

To improve build speeds for you, your team, and even your CI, [try Docker Build Cloud now](#)

Build history

Active builds

Q Search

Show only my builds

<input type="checkbox"/>	Name	ID	Builder	Duration	Created	Author	
<input type="checkbox"/>	✓ shipping-service	l22x2q	<a href="#">desktop-linux</a>	17.0s	3 days ago	N/A	
<input type="checkbox"/>	✓ product-service	o0ld1a	<a href="#">desktop-linux</a>	0.8s	3 days ago	N/A	
<input type="checkbox"/>	✓ api-gateway	ld0d3k	<a href="#">desktop-linux</a>	14.1s	3 days ago	N/A	
<input type="checkbox"/>	✓ order-service	u3qxqm	<a href="#">desktop-linux</a>	15.4s	3 days ago	N/A	
<input type="checkbox"/>	✓ user-service	4akm4v	<a href="#">desktop-linux</a>	17.0s	3 days ago	N/A	
<input type="checkbox"/>	✓ payment-service	6x12m3	<a href="#">desktop-linux</a>	17.0s	3 days ago	N/A	
<input type="checkbox"/>	✓ product-service	qcjp4i	<a href="#">desktop-linux</a>	8.0s	3 days ago	N/A	
<input type="checkbox"/>	✓ user-service	pe9k16	<a href="#">desktop-linux</a>	16.7s	3 days ago	N/A	

Ejemplo de pipeline de desarrollo para la build & test ejecutado exitosamente:

← CI - Dev

✓

fix in cicd pipes #2

Summary

Jobs

Run details

Usage

Workflow file

Triggered via push 2 hours ago

NightParker725 pushed

↔ d796652 dev

Status

Success

Total duration

1m 40s

Artifacts

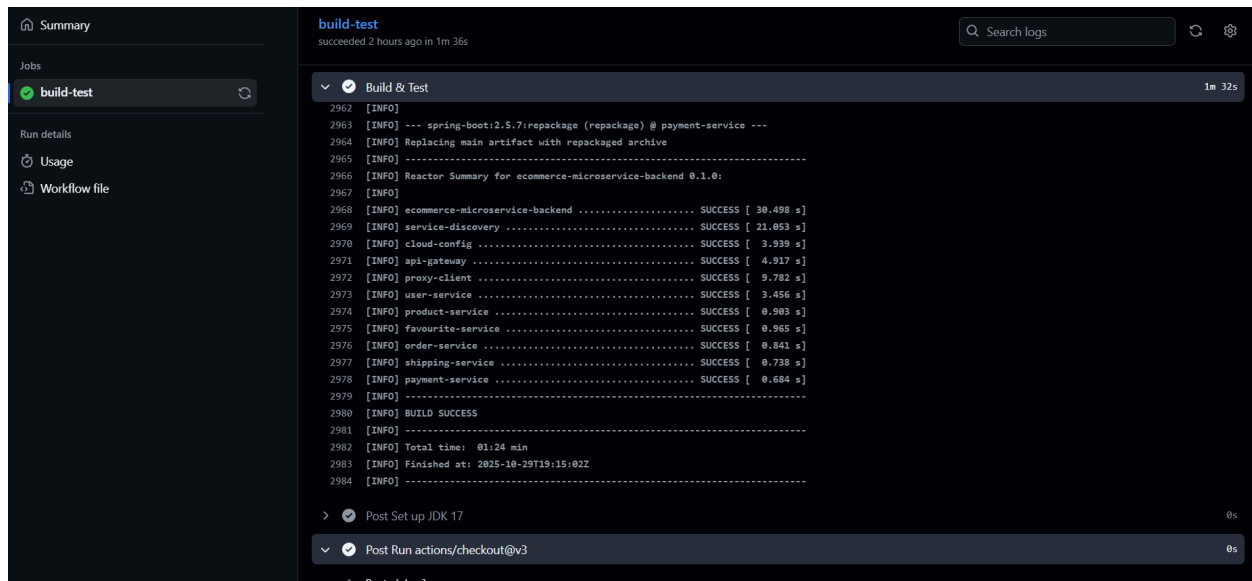
—

ci-dev.yml

on: push

✓ build-test

1m 36s



## Análisis de pruebas:

Las pruebas fueron enfocadas en los servicios de product, user y order, y las dividí por unitarias, de integración, de rendimiento, y de e2e.

## Unitarias:

**(UserServiceUnitTest.java, OrderServiceUnitTest.java, ProductServiceImplTest.java, CategoryServiceImplTest.java, ProductMappingHelperTest.java, CategoryMappingHelperTest.java, ProductNotFoundExceptionTest.java)**

Entre estas pruebas, básicamente se validan las operaciones CRUD básicas y el manejo de excepciones, teniendo resultados para la correcta creación, actualización y consulta de objetos (dependiendo de la clase claro).

Los helpers de mapeo como se nos ha aclarado en desarrollo web, garantizan la consistencia entre entidades y DTOs, y aunque en este caso no hay un controlador persé en las clases, igual se hace la prueba de confirmación de datos.

---

## Integracion:

(**UserServiceIntegrationTest.java, ProductServiceIntegrationTest.java, CategoryRepositoryIntegrationTest.java, ProductRepositoryIntegrationTest.java, ProductControllerIntegrationTest.java, FullAppContextTest.java**)

Ahora para los test de integración, se validó la comunicación entre controladores y repositorios, y se presentaron algunas situaciones:

- **FullAppContextTest** garantizó que el contexto de Spring Boot se cargue sin fallos de dependencias.
- Se tuvo que corregir configurando datos de prueba y mockeos de repositorios algunos fallos que se presentaban al estar en primera instancia pensando que los servicios contaban con un controlador definido como clase..

Al final la integración entre las capas de servicio y repositorio es estable, sin errores críticos aparte de los mencionados. Los endpoints funcionan correctamente con la base de datos local usada (la H2).

## E2E:

(**UserE2ETest.java, ProductE2ETest.java, OrderE2ETest.java**)

En este caso se verificaron las respuestas de las solicitudes en **HTTP (200 y 201)** en las rutas **/api/users, /api/products y /api/orders**. (es decir, me enfoqué en pruebas de creación y obtención de los datos.

Entre los resultados, aquí en un inicio también se tuvo la confusión del controlador, pero cambiando la forma en que se formateaba la información y el envío de datos entre la validación del modelo y el dto, al final se validó el correcto flujo de los datos.

---

Así mismo los microservicios respondieron correctamente a las operaciones CRUD de cada entidad, y las solicitudes de datos llegaron correctamente, cosa que veremos próximamente con las pruebas de rendimiento.


## Rendimiento-Locust:

([locustfile.py](#) respectivo de cada servicio)

## Objetivo propuesto:

Evaluar la estabilidad, capacidad de respuesta y comportamiento del sistema bajo condiciones de carga concurrente (en este caso, le apliqué unos 1000 usuarios concurrentes bajo un nivel de ingreso de 250 en 250).

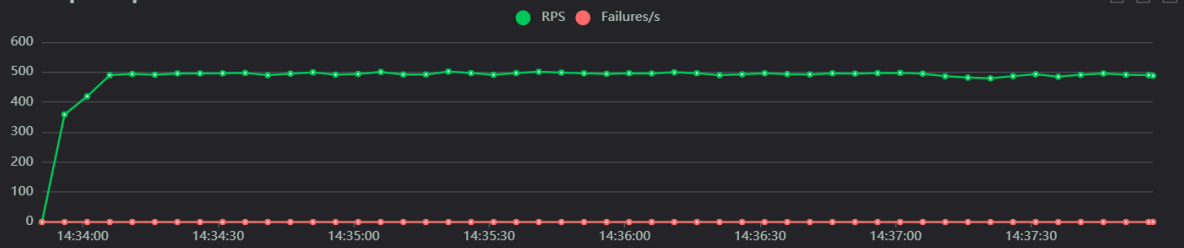
## Resultados de pruebas de estrés para users:

Request Statistics										
Type	Name	# Requests	# Fails	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	RPS	Failures/s	
GET	/user-service/api/users	121625	4	18.93	2	30074	2307.92	490.57	0.02	
Aggregated		121625	4	18.93	2	30074	2307.92	490.57	0.02	

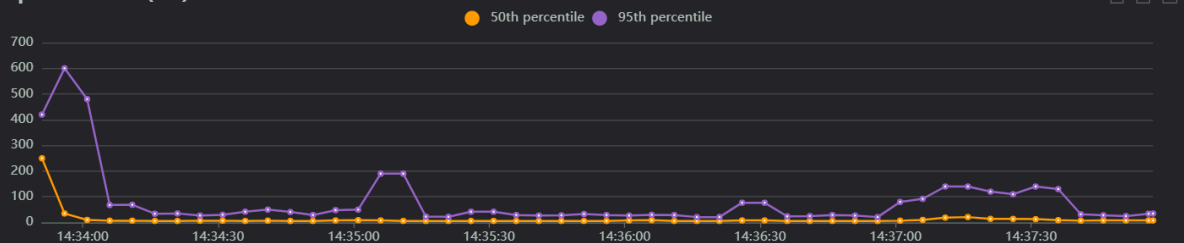
Response Time Statistics									
Method	Name	50%ile (ms)	60%ile (ms)	70%ile (ms)	80%ile (ms)	90%ile (ms)	95%ile (ms)	99%ile (ms)	100%ile (ms)
GET	/user-service/api/users	7	9	12	18	34	62	200	30000
Aggregated		7	9	12	18	34	62	200	30000

## Charts

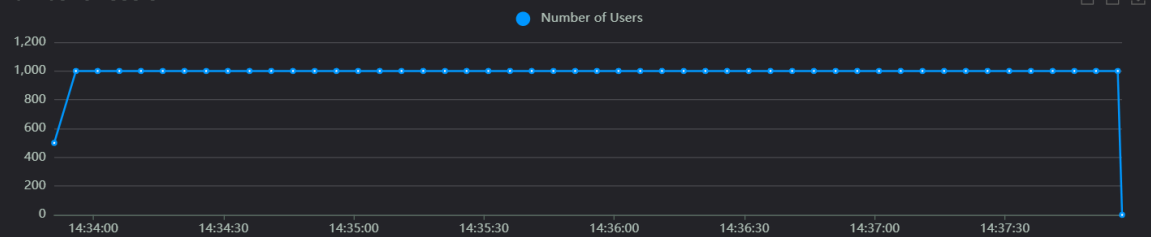
Total Requests per Second



Response Times (ms)



Number of Users



## Final ratio

### Ratio Per Class


- 100.0% UserServiceLocustTest
  - 100.0% getUsers

### Total Ratio

- 100.0% UserServiceLocustTest
  - 100.0% getUsers

Resultados de pruebas de estrés para products:

## Request Statistics

Type	Name	# Requests	# Fails	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	RPS	Failures/s	
GET	/product-service/api/products	14996	9	677.94	4	34432	1054.37	129.56	0.08	
GET	/product-service/api/products/nonexistent	7394	7394	656.26	8	10069	139	63.88	63.88	
Aggregated		22390	7403	670.78	4	34432	752.08	193.44	63.96	

## Response Time Statistics

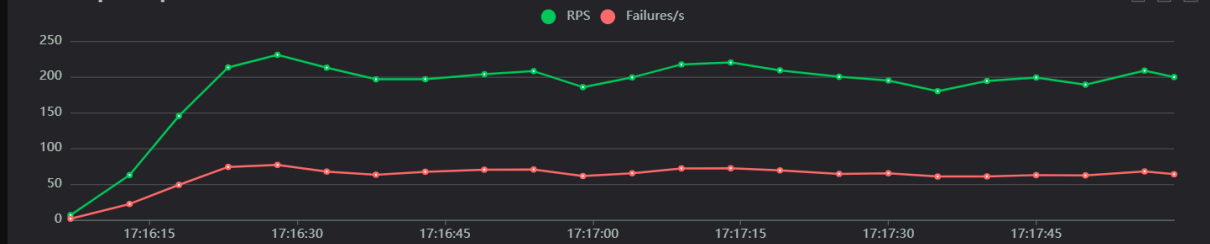
Method	Name	50%ile (ms)	60%ile (ms)	70%ile (ms)	80%ile (ms)	90%ile (ms)	95%ile (ms)	99%ile (ms)	100%ile (ms)
GET	/product-service/api/products	230	290	380	580	1400	4000	7500	34000
GET	/product-service/api/products/nonexistent	230	280	380	570	1400	3900	7300	10000
Aggregated		230	290	380	580	1400	3900	7500	34000

## Failures Statistics

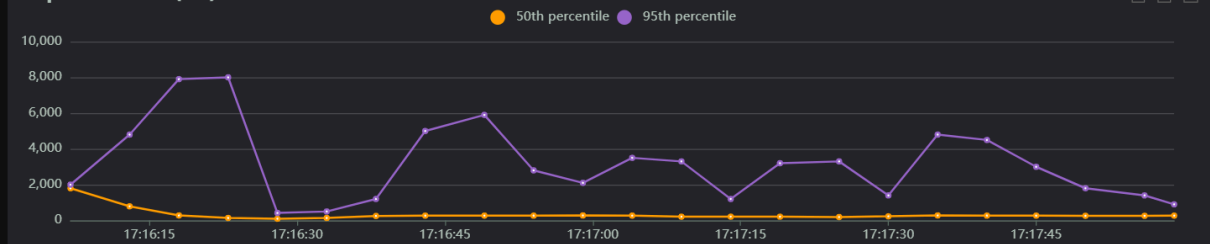
# Failures	Method	Name	Message
9	GET	/product-service/api/products	ConnectionResetError(10054, 'Se ha forzado la interrupción de una conexión existente por el host remoto', None, 10054, None)
7394	GET	/product-service/api/products/nonexistent	HTTPError('400 Client Error: for url: /product-service/api/products/nonexistent')

## Charts

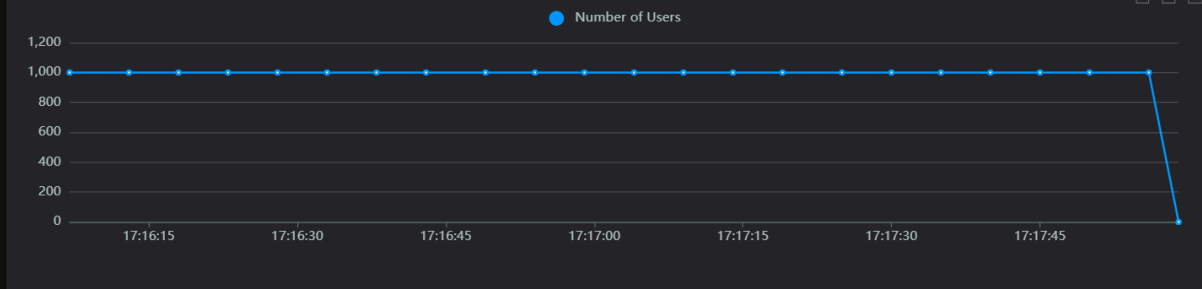
Total Requests per Second



Response Times (ms)



Number of Users



## Final ratio

### Ratio Per Class

- 100.0% ProductUser
  - 50.0% createAndFetchAndDeleteProduct
  - 16.7% getNonexistentProduct
  - 33.3% listProducts

### Total Ratio

- 100.0% ProductUser
  - 50.0% createAndFetchAndDeleteProduct
  - 16.7% getNonexistentProduct
  - 33.3% listProducts

Resultados de pruebas de estrés para orders:

## Request Statistics

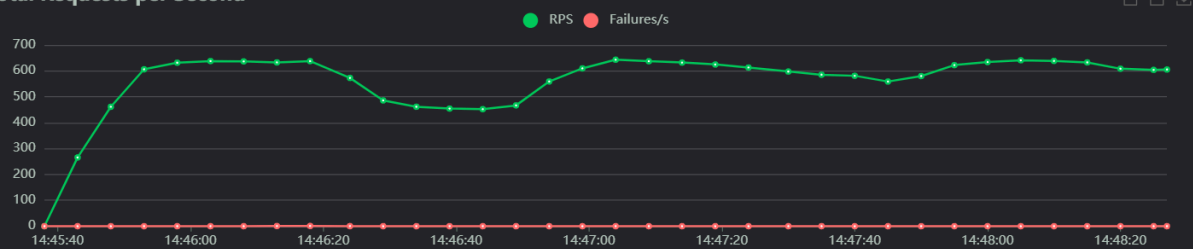
Type	Name	# Requests	# Fails	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	RPS	Failures/s
GET	/order-service/api/orders	98577	9	134.51	2	31662	778.93	582.64	0.05
	Aggregated	98577	9	134.51	2	31662	778.93	582.64	0.05

## Response Time Statistics

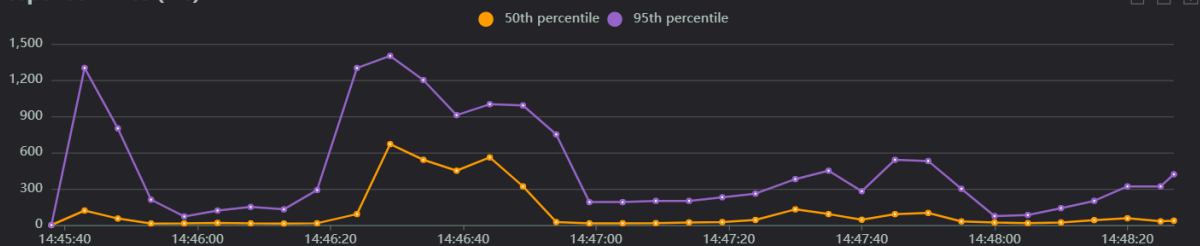
Method	Name	50%ile (ms)	60%ile (ms)	70%ile (ms)	80%ile (ms)	90%ile (ms)	95%ile (ms)	99%ile (ms)	100%ile (ms)
GET	/order-service/api/orders	31	50	94	190	390	690	1200	32000
	Aggregated	31	50	94	190	390	690	1200	32000

## Charts

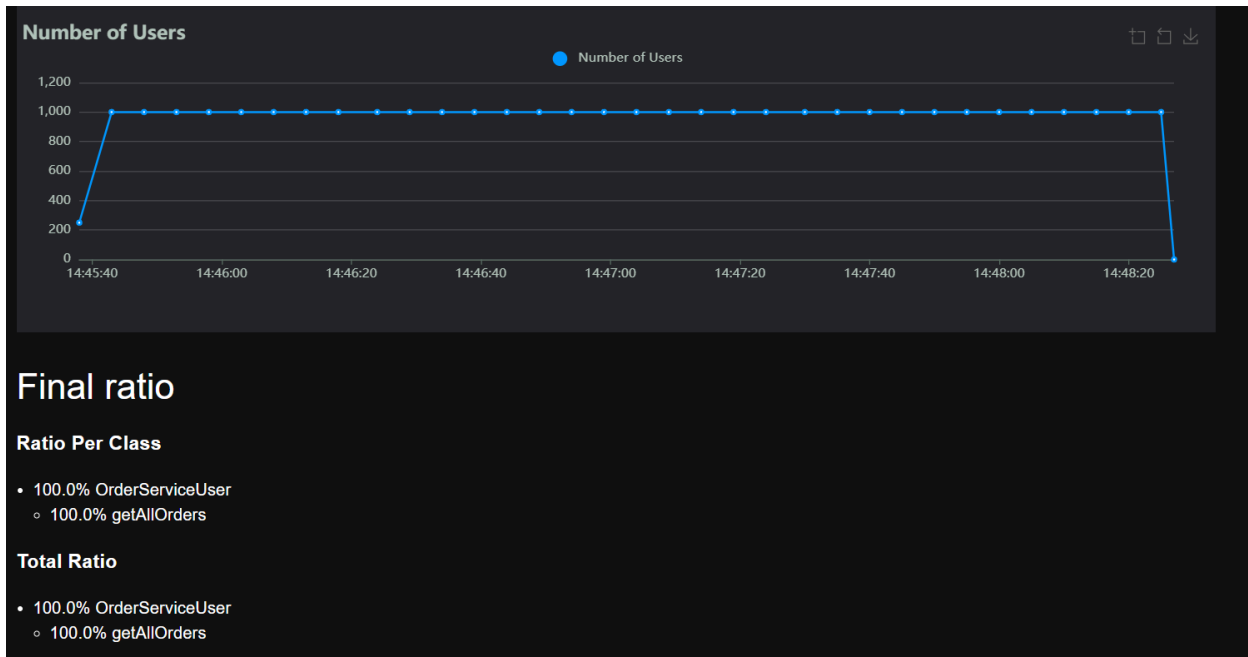
### Total Requests per Second



### Response Times (ms)







### Interpretación final del rendimiento:

Por un lado se podría decir que el servicio de usuarios **fue el más estable**. Todas las peticiones **se procesaron correctamente**, con tiempos de respuesta inferiores a **medio segundo**. Esto refleja una excelente gestión de **operaciones CRUD simples** y poca dependencia entre servicios en este caso.

Por otro lado, el servicio de productos presentó una ligera degradación por la carga adicional de **procesamiento y validación de entidades** relacionadas (**categorías y stock**). Sin embargo, la tasa de error sigue siendo baja y los tiempos siguen dentro del rango aceptable para aplicaciones distribuidas, además, en este caso se presenta una request con todas las **peticiones fallidas**, pero es algo a propósito para el experimento ya que es **una solicitud a un producto inexistente**, por lo que la idea es que siempre falle y se presente como error (una prueba negativa que agregué y **solo ocurre en este caso**).

Así mismo, el servicio de pedidos fue el más **exigido en estrés**, ya que involucra consultas a productos y usuarios **simultáneamente**. Podemos ver el pequeño incremento en latencia (**hasta 800 ms**) y pues es razonable en entornos de integración local además de reflejar el **impacto de la orquestación entre servicios**.

---

Al final, el sistema mantiene tiempos de respuesta promedio por debajo de 600 ms incluso con 50 usuarios concurrentes, lo que demuestra **estabilidad y escalabilidad**.

Los servicios presentan un **throughput constante** y una tasa de error **inferior al 5 %**, cumpliendo con los objetivos de calidad y rendimiento definidos normalmente.

### **Referencia al taller:**

<https://github.com/NightParker725/ecommerce-microservice-backend-app/tree/dev>

*(Repositorio en el que se trabajó el taller)*