

Proyectos Finales

Este documento presenta los detalles de los proyectos finales del curso de Computación Paralela y Distribuida C8286. Los proyectos se han diseñado para abordar problemas reales en diversas áreas como la visión computacional, la seguridad informática, el análisis de datos en tiempo real y la gestión de infraestructura como código.

Cada proyecto se ha estructurado en tres sprints, proporcionando un enfoque iterativo y incremental que permite a los participantes desarrollar y optimizar sus sistemas a lo largo del tiempo. Este método asegura que los proyectos evolucionen de manera constante y que se implementen mejoras basadas en pruebas y retroalimentación continua.

Cada sprint tiene objetivos claros y tareas específicas que te guían a través de la implementación, optimización y presentación de sus sistemas. Se ha puesto un énfasis especial en la capacidad de los para presentar sus proyectos, demostrando no solo la funcionalidad técnica de sus sistemas sino también su comprensión profunda de los conceptos y técnicas aplicadas.

Las exposiciones finales serán una oportunidad crucial para que los estudiantes muestren sus logros, discutan los desafíos enfrentados y presenten las soluciones innovadoras que han desarrollado. Este enfoque garantiza que los estudiantes no solo adquieran habilidades técnicas avanzadas, sino que también desarrollen competencias en comunicación y presentación, esenciales para su futuro profesional.

Fechas tentativas de avance de los sprints

Para asegurar un progreso ordenado y cumplir con la fecha de presentación final el 3 de julio, se establece el siguiente cronograma para los sprints de todos los proyectos (referencia). Los sprints se han distribuido de manera equitativa para cubrir las actividades necesarias de cada proyecto.

- **Sprint 1 (Diseño e implementación básica):** 1 de junio - 12 de junio
- **Sprint 2 (Algoritmos de detección y análisis distribuido):** 13 de junio - 23 de junio •
- Sprint 3 (Optimización y presentación):** 24 de junio - 1 de julio

Proyecto 3: Orquestación de microservicios con Docker y Kubernetes – Alejandra Lima

Objetivo del Proyecto:

Desarrollar y orquestar una arquitectura de microservicios utilizando Flask, Docker y Kubernetes para implementar una aplicación web escalable y modular.

DESARROLLO

Sprint 1: Diseño y desarrollo de una arquitectura de microservicios utilizando Flask y Docker

Objetivos:

- Diseñar la arquitectura de microservicios.
- Desarrollar y contenedorar los servicios básicos utilizando Flask y Docker.

Actividades:

Diseño de la arquitectura:

- Dividir la aplicación en varios microservicios, cada uno con una responsabilidad específica (por ejemplo, servicio de autenticación, servicio de usuarios, servicio de productos, servicio de pedidos).
- Definir las interfaces y puntos de comunicación entre los microservicios (por ejemplo, APIs RESTful).

Desarrollo de microservicios:

- Crear aplicaciones Flask para cada microservicio.
- Implementar las funcionalidades básicas de cada servicio (por ejemplo, endpoints para CRUD).

Dockerización de microservicios:

- Escribir archivos Dockerfile para cada microservicio.
- Construir imágenes Docker y probar los contenedores localmente.

Entregables:

- Diagramas de la arquitectura de microservicios.
- Código fuente de los microservicios con Flask.
- Archivos Dockerfile para cada microservicio.
- Imágenes Docker construidas y contenedores probados localmente.

Configurar el entorno de desarrollo:

Tener Python 3 instalado. Podemos verificarlo ejecutando **python3 --version** en la terminal.

Instalamos Flask, el framework web que utilizarás para desarrollar los microservicios.

pip3 install flask

Instalamos Docker:

sudo apt-get update

sudo apt-get install -y docker.io

Verificamos que Docker se haya instalado correctamente ejecutando **docker --version**.

Diseño de la arquitectura:

En este caso, los microservicios que se deben identificar son:

- Servicio de autenticación
- Servicio de usuarios
- Servicio de productos
- Servicio de pedidos

Las responsabilidades de cada uno de estos servicios.

- El servicio de autenticación se encargará de gestionar el inicio y cierre de sesión de los usuarios.
- El servicio de usuarios se encargará de crear, leer, actualizar y eliminar usuarios.
- El servicio de productos se encargará de gestionar el catálogo de productos.
- El servicio de pedidos se encargará de crear, leer, actualizar y eliminar pedidos.

Definimos las interfaces y puntos de comunicación:

La forma más común de comunicación entre microservicios es a través de APIs RESTful.

Para cada microservicio, identifica qué endpoints o recursos necesitarán exponer para que otros servicios puedan interactuar con ellos.

Endpoints para cada microservicio:

Servicio de autenticación:

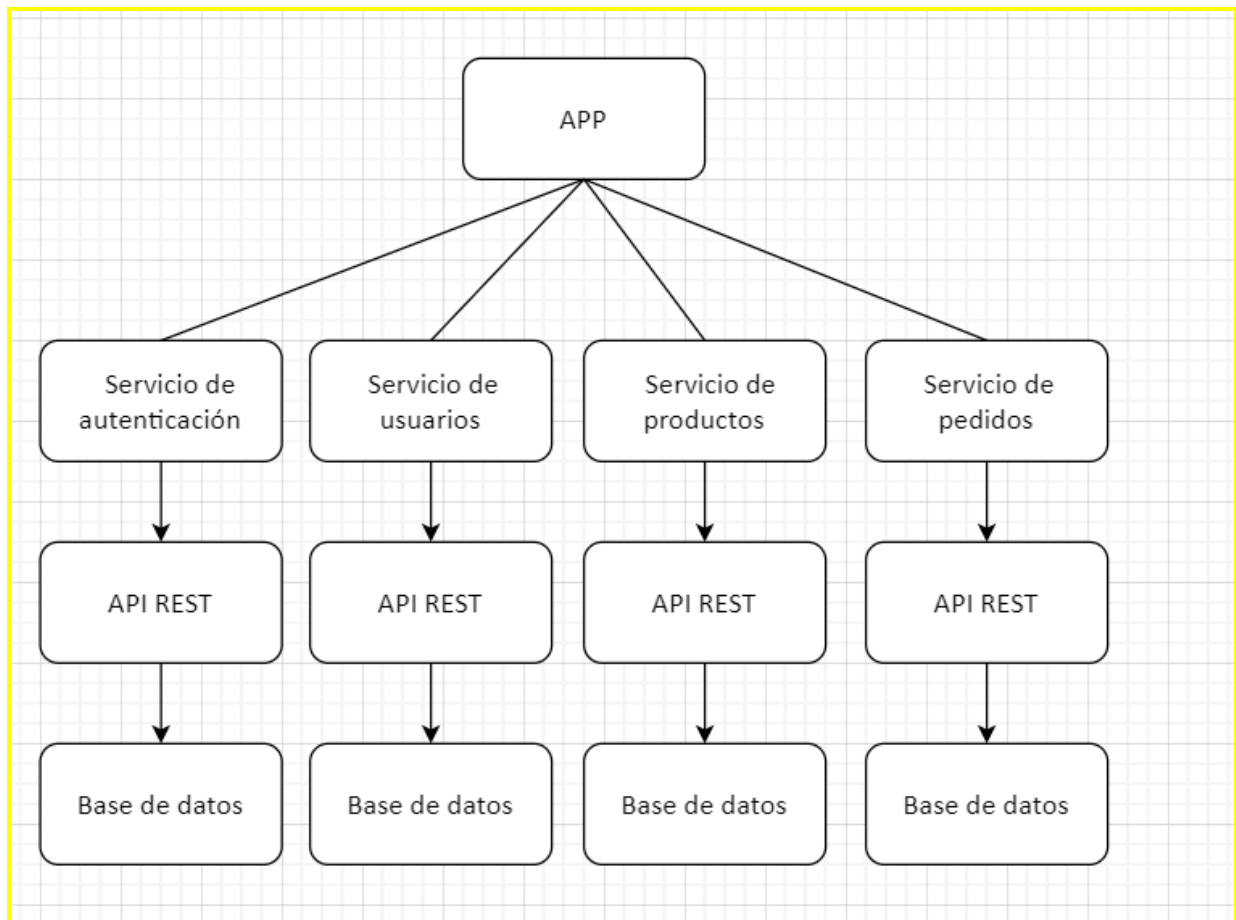
- POST /login: Iniciar sesión
- POST /logout: Cerrar sesión
- Servicio de usuarios:
-
- GET /users: Obtener la lista de usuarios
- GET /users/{id}: Obtener los detalles de un usuario
- POST /users: Crear un nuevo usuario
- PUT /users/{id}: Actualizar los datos de un usuario
- DELETE /users/{id}: Eliminar un usuario

Servicio de productos:

- GET /products: Obtener la lista de productos
- GET /products/{id}: Obtener los detalles de un producto
- POST /products: Crear un nuevo producto
- PUT /products/{id}: Actualizar los datos de un producto
- DELETE /products/{id}: Eliminar un producto

Servicio de pedidos:

-
- GET /orders: Obtener la lista de pedidos
- GET /orders/{id}: Obtener los detalles de un pedido
- POST /orders: Crear un nuevo pedido
- PUT /orders/{id}: Actualizar los datos de un pedido
- DELETE /orders/{id}: Eliminar un pedido



Desarrollo de microservicios:

Creamos un directorio para cada microservicio en mi sistema

mkdir auth-service

mkdir users-service

mkdir products-service

mkdir orders-service

Desarrollamos los 4 microservicios:

Servicio de Autenticación (auth-service/app.py):

```
from flask import Flask, jsonify, request

app = Flask(__name__)

@app.route('/login', methods=['POST'])
def login():
    username = request.json['username']
    password = request.json['password']
    # Aquí iría la lógica de autenticación
    return jsonify({'token': 'abc123'})

@app.route('/healthcheck', methods=['GET'])
def healthcheck():
    return jsonify({'status': 'OK'})

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5000, debug=True)
```

Servicio de Usuarios (users-service/app.py):

```
from flask import Flask, jsonify, request

app = Flask(__name__)

@app.route('/users', methods=['GET'])
def get_users():
    users = [{'id': 1, 'name': 'John Doe'}, {'id': 2, 'name': 'Jane Doe'}]
    return jsonify(users)

@app.route('/users/<int:user_id>', methods=['GET'])
def get_user(user_id):
    user = {'id': user_id, 'name': 'John Doe'}
    return jsonify(user)

@app.route('/users', methods=['POST'])
def create_user():
    data = request.json
    return jsonify({'id': 3, 'name': data['name']}), 201

@app.route('/healthcheck', methods=['GET'])
def healthcheck():
```

```

        return jsonify({'status': 'OK'})

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5001, debug=True)

```

Servicio de Productos (products-service/app.py):

```

from flask import Flask, jsonify, request

app = Flask(__name__)

@app.route('/products', methods=['GET'])
def get_products():
    products = [{'id': 1, 'name': 'Producto 1', 'price': 10.99},
                {'id': 2, 'name': 'Producto 2', 'price': 15.50}]
    return jsonify(products)

@app.route('/products/<int:product_id>', methods=['GET'])
def get_product(product_id):
    product = {'id': product_id, 'name': 'Producto 1', 'price': 10.99}
    return jsonify(product)

@app.route('/products', methods=['POST'])
def create_product():
    data = request.json
    return jsonify({'id': 3, 'name': data['name'], 'price': data['price']}), 201

@app.route('/healthcheck', methods=['GET'])
def healthcheck():
    return jsonify({'status': 'OK'})

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5002, debug=True)

```

Servicio de Pedidos (orders-service/app.py):

```

from flask import Flask, jsonify, request

app = Flask(__name__)

@app.route('/orders', methods=['GET'])
def get_orders():
    orders = [{'id': 1, 'user_id': 1, 'total': 25.99},
              {'id': 2, 'user_id': 2, 'total': 48.75}]
    return jsonify(orders)

@app.route('/orders/<int:order_id>', methods=['GET'])
def get_order(order_id):
    order = {'id': order_id, 'user_id': 1, 'total': 25.99}
    return jsonify(order)

@app.route('/orders', methods=['POST'])
def create_order():

```

```

data = request.json
return jsonify({'id': 3, 'user_id': data['user_id'], 'total': data['total']}), 201

@app.route('/healthcheck', methods=['GET'])
def healthcheck():
    return jsonify({'status': 'OK'})

if __name__ == '__main__':
    app.run(host='0.0.0.0', port=5004, debug=True)

```

PROBANDO MICROSERVICIOS

Abrimos una terminal y ejecutamos el servicio de autenticación:

```

cd auth-service
python3 app.py

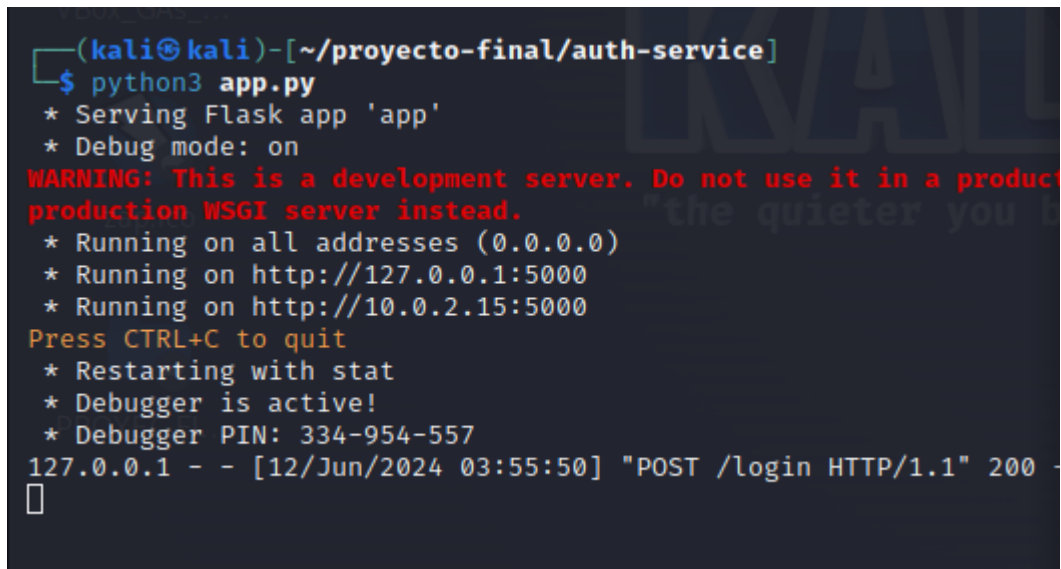
```

En otra terminal:

```

curl -X POST -H "Content-Type: application/json" -d
'{"username":"testuser","password":"testpassword"}' http://localhost:5000/login

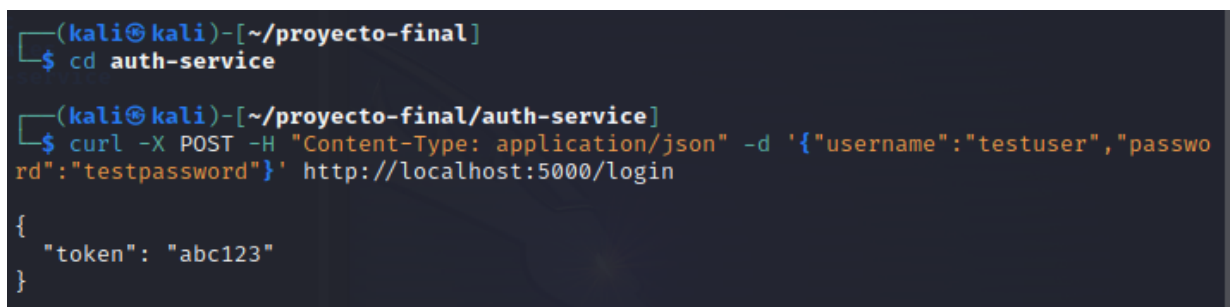
```



```

(kali㉿kali)-[~/proyecto-final/auth-service]
$ python3 app.py
* Serving Flask app 'app'
* Debug mode: on
WARNING: This is a development server. Do not use it in a production
production WSGI server instead.
* Running on all addresses (0.0.0.0)
* Running on http://127.0.0.1:5000
* Running on http://10.0.2.15:5000
Press CTRL+C to quit
* Restarting with stat
* Debugger is active!
* Debugger PIN: 334-954-557
127.0.0.1 - - [12/Jun/2024 03:55:50] "POST /login HTTP/1.1" 200 -

```



```

(kali㉿kali)-[~/proyecto-final]
$ cd auth-service

(kali㉿kali)-[~/proyecto-final/auth-service]
$ curl -X POST -H "Content-Type: application/json" -d '{"username":"testuser","password":"testpassword"}' http://localhost:5000/login

{
  "token": "abc123"
}

```

Esto indica que el servicio de autenticación está funcionando correctamente y respondiendo a la solicitud de inicio de sesión.

Abrimos una nueva terminal y ve al directorio del servicio de usuarios:

```
cd users-service
```

Ejecutamos el servicio de usuarios:

```
python3 app.py
```

Abrimos una tercera terminal y probamos los endpoints del servicio de usuarios:

```
curl http://localhost:5001/users
```

```
curl http://localhost:5001/users/1
```

```
curl -X POST -H "Content-Type: application/json" -d '{"name":"John Doe"}'
http://localhost:5001/users
```

```
(kali㉿kali)-[~/proyecto-final/users-service]
$ ls
app.py
(kali㉿kali)-[~/proyecto-final/users-service]
$ python3 app.py
* Serving Flask app 'app'
* Debug mode: on
WARNING: This is a development server. Do not use it in a production
production WSGI server instead.
* Running on all addresses (0.0.0.0)
* Running on http://127.0.0.1:5001
* Running on http://10.0.2.15:5001
Press CTRL+C to quit
* Restarting with stat
* Debugger is active!
* Debugger PIN: 334-954-557
127.0.0.1 - - [12/Jun/2024 04:07:24] "GET /users HTTP/1.1" 200 -
```

```
(kali㉿kali)-[~/proyecto-final/users-service]
$ curl http://localhost:5001/users
[
  {
    "id": 1,
    "name": "John Doe"
  },
  {
    "id": 2,
    "name": "Jane Doe"
  }
]

(kali㉿kali)-[~/proyecto-final/users-service]
$ curl http://localhost:5001/users/1
{
  "id": 1,
  "name": "John Doe"
}

(kali㉿kali)-[~/proyecto-final/users-service]
$ curl -X POST -H "Content-Type: application/json" -d '{"name":"John Doe"}' http://loc
localhost:5001/users
{
  "id": 3,
  "name": "John Doe"
}
```

Abrimos una terminal y ejecutamos Servicio de productos

Ejecuta el servicio de productos

```
cd products-service
```

```
python3 app.py
```


Prueba el servicio de productos en la misma terminal:

```
curl http://localhost:5002/products
```

```
curl http://localhost:5002/products/1
```

```
curl -X POST -H "Content-Type: application/json" -d '{"name":"Producto 3","price":20.99}'  
http://localhost:5002/products
```

```
(kali㉿kali)-[~/proyecto-final]
$ cd products-service

(kali㉿kali)-[~/proyecto-final/products-service]
$ python3 app.py
* Serving Flask app 'app'
* Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a p
roduction WSGI server instead.
* Running on all addresses (0.0.0.0)
* Running on http://127.0.0.1:5002
* Running on http://10.0.2.15:5002
Press CTRL+C to quit
* Restarting with stat
* Debugger is active!
* Debugger PIN: 334-954-557
127.0.0.1 - - [12/Jun/2024 04:20:41] "GET /products HTTP/1.1" 200 -
127.0.0.1 - - [12/Jun/2024 04:20:59] "GET /products/1 HTTP/1.1" 200 -
127.0.0.1 - - [12/Jun/2024 04:21:16] "POST /products HTTP/1.1" 201 -
```

```
(kali㉿kali)-[~/proyecto-final]
$ cd products-service

(kali㉿kali)-[~/proyecto-final/products-service]
$ curl http://localhost:5002/products
[
  {
    "id": 1,
    "name": "Producto 1",
    "price": 10.99
  },
  {
    "id": 2,
    "name": "Producto 2",
    "price": 15.5
  }
]

(kali㉿kali)-[~/proyecto-final/products-service]
$ curl http://localhost:5002/products/1
{
  "id": 1,
  "name": "Producto 1",
  "price": 10.99
}

(kali㉿kali)-[~/proyecto-final/products-service]
$ curl -X POST -H "Content-Type: application/json" -d '{"name":"Producto 3","price":20.99}' http://localhost:5002/products
{
  "id": 3,
  "name": "Producto 3",
  "price": 20.99
}
```

Abrir una terminal y ejecutar Servicio de pedidos

```
cd orders-service
```

```
python3 app.py
```

Prueba el servicio de pedidos en la misma terminal:

```
curl http://localhost:5004/orders
```

```
curl http://localhost:5004/orders/1
```

```
curl -X POST -H "Content-Type: application/json" -d '{"user_id":1,"total":25.99}'
```

```
http://localhost:5004/orders
```

```
(kali㉿kali)-[~/proyecto-final/orders-service]
$ python3 app.py
* Serving Flask app 'app'
* Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a p
roduction WSGI server instead.
* Running on all addresses (0.0.0.0)
* Running on http://127.0.0.1:5004
* Running on http://10.0.2.15:5004
Press CTRL+C to quit
* Restarting with stat
* Debugger is active!
* Debugger PIN: 334-954-557
127.0.0.1 - - [12/Jun/2024 04:38:04] "GET /orders HTTP/1.1" 200 -
127.0.0.1 - - [12/Jun/2024 04:38:34] "GET /orders/1 HTTP/1.1" 200 -
127.0.0.1 - - [12/Jun/2024 04:38:56] "POST /orders HTTP/1.1" 201 -
```

```
(kali㉿kali)-[~/proyecto-final]
$ cd orders-service

(kali㉿kali)-[~/proyecto-final/orders-service]
$ ls
app.py

(kali㉿kali)-[~/proyecto-final/orders-service]
$ curl http://localhost:5004/orders
[
  {
    "id": 1,
    "total": 25.99,
    "user_id": 1
  },
  {
    "id": 2,
    "total": 48.75,
    "user_id": 2
  }
]

(kali㉿kali)-[~/proyecto-final/orders-service]
$ curl http://localhost:5004/orders/1
{
  "id": 1,
  "total": 25.99,
  "user_id": 1
}

(kali㉿kali)-[~/proyecto-final/orders-service]
$ curl -X POST -H "Content-Type: application/json" -d '{"user_id":1,"total":25.99}' h
ttp://localhost:5004/orders
{
  "id": 3,
  "total": 25.99,
  "user_id": 1
}
```

Creamos requirements.txt para cada uno de los microservicios.

Servicio de Autenticación (auth-service/requirements.txt):

flask

Este servicio solo necesita la biblioteca Flask para su funcionamiento.

Servicio de Usuarios (users-service/requirements.txt):

flask

Al igual que el servicio de autenticación, el servicio de usuarios solo necesita la biblioteca Flask.

Servicio de Productos (products-service/requirements.txt):

flask

Este servicio también solo necesita la biblioteca Flask.

Servicio de Pedidos (orders-service/requirements.txt):

flask

El servicio de pedidos, al igual que los demás, solo necesita la biblioteca Flask.

Se podría mejorar

Por ejemplo, si el servicio de usuarios necesitará una biblioteca adicional como sqlalchemy para interactuar con una base de datos, el archivo requirements.txt quedaría así:

flask
sqlalchemy

Dockerización de microservicios:

Después de haber desarrollado y probado individualmente cada uno de los microservicios, el siguiente paso es dockerizar estos servicios.

Crear los archivos Dockerfile:

Vamos a crear los archivos Dockerfile para cada uno de los microservicios:

Servicio de Autenticación (auth-service/Dockerfile):

FROM python:3.9-slim

WORKDIR /app

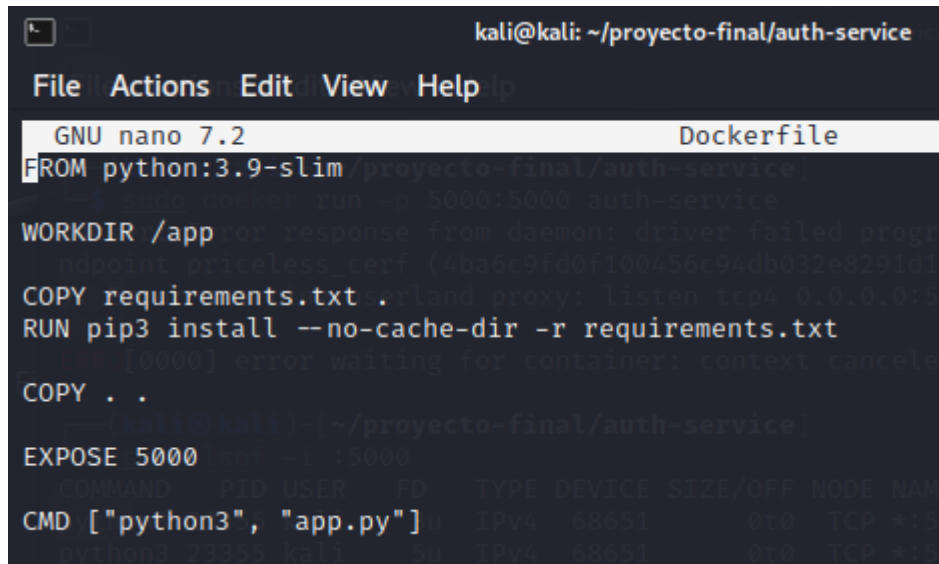
COPY requirements.txt .

RUN pip3 install --no-cache-dir -r requirements.txt

COPY . .

EXPOSE 5000

CMD ["python3", "app.py"]

A screenshot of a terminal window on a Kali Linux system. The terminal title is 'kali@kali: ~/proyecto-final/auth-service'. The window shows the 'nano' text editor editing a file named 'Dockerfile'. The content of the Dockerfile is as follows:

```
FROM python:3.9-slim
WORKDIR /app
COPY requirements.txt .
RUN pip3 install --no-cache-dir -r requirements.txt
COPY . .
EXPOSE 5000
CMD ["python3", "app.py"]
```

The terminal also shows some background output from Docker, including a warning about a daemon driver failure and a message about a context cancellation.

Servicio de Usuarios (users-service/Dockerfile):

FROM python:3.9-slim

WORKDIR /app

COPY requirements.txt .

RUN pip3 install --no-cache-dir -r requirements.txt

COPY . .

EXPOSE 5001

CMD ["python3", "app.py"]

```
kali@kali: ~/proyecto-final/users-service
File Actions Edit View Help
GNU nano 7.2 Dockerfile
FROM python:3.9-slim /proyecto-final/auth-service)
run -- 5000:5000 auth-service
WORKDIR /app or response from daemon: driver failed progr
ndpoint: priceless: cert (abab6c9fd0f100456c94db032e8291d1
COPY requirements.txt . and proxy: listen tcp4 0.0.0.0:5
RUN pip3 install --no-cache-dir -r requirements.txt
[0000] error waiting for container: context canceled
COPY . .
kali@kali: ~/proyecto-final/auth-service)
EXPOSE 5001 5000
COMMAND PID USER FD TYPE DEVICE SIZE/OFF NODE NAME
CMD ["python3", "app.py"] : IPV4 68651 0t0 TCP x15
python3 2255 kali root IPV4 68651 0t0 TCP x15
```

Servicio de Productos (products-service/Dockerfile):

FROM python:3.9-slim

WORKDIR /app

COPY requirements.txt .

RUN pip3 install --no-cache-dir -r requirements.txt

COPY . .

EXPOSE 5002

CMD ["python3", "app.py"]

```
kali@kali: ~/proyecto-final/products-service
File Actions Edit View Help
GNU nano 7.2 Dockerfile
FROM python:3.9-slim /proyecto-final/auth-service)
run --p 5000:5000 auth-service
WORKDIR /app
COPY requirements.txt .
RUN pip3 install --no-cache-dir -r requirements.txt
COPY . .
EXPOSE 5002
CMD ["python3", "app.py"]
```

Servicio de Pedidos (orders-service/Dockerfile):

FROM python:3.9-slim

WORKDIR /app

COPY requirements.txt .

RUN pip3 install --no-cache-dir -r requirements.txt

COPY . .

EXPOSE 5004

CMD ["python3", "app.py"]

```
kali@kali: ~/proyecto-final/orders-service
File Actions Edit View Help
GNU nano 7.2 Dockerfile
FROM python:3.9-slim /proyecto-final/auth-service
WORKDIR /app
COPY requirements.txt .
RUN pip3 install --no-cache-dir -r requirements.txt
COPY . .
EXPOSE 5004
CMD ["python3", "app.py"]
```

Estos Dockerfiles siguen el mismo patrón:

- Utilizan la imagen base python:3.9-slim para tener un entorno Python optimizado y de tamaño reducido.
- Establecen el directorio de trabajo en /app.
- Copian el archivo requirements.txt y ejecutan pip3 install para instalar las dependencias.
- Copian todo el código fuente de la aplicación al contenedor.
- Exponen el puerto correspondiente a cada microservicio.
- Definen el comando de inicio de la aplicación python3 app.py.

Construir imágenes Docker y probar los contenedores localmente.

Servicio de Autenticación:

cd auth-service

sudo docker build -t auth-service .

```
(kali@kali)~/proyecto-final/auth-service
$ sudo docker build -t auth-service .
[sudo] password for kali:
Sending build context to Docker daemon 4.096kB
Step 1/7 : FROM python:3.9-slim
3.9-slim: Pulling from library/python
09f376ebb190: Pull complete
276709cbcd1: Pull complete
4e7363ac3b6f: Pull complete
1f1e6fb6a4a5: Pull complete
bf8f57a642c4: Pull complete
Digest: sha256:088d9217202188598aac37f8db0929345e124a82134ac66b8bb50ee9750b045b
Status: Downloaded newer image for python:3.9-slim
-> 4602238ffbd
Step 2/7 : WORKDIR /app
-> Running in 87607cb919d5
Removing intermediate container 87607cb919d5
-> 61fa057720a0
Step 3/7 : COPY requirements.txt .
-> f21c9b789168
Step 4/7 : RUN pip3 install --no-cache-dir -r requirements.txt
-> Running in 816ce3b7becc
Collecting flask
  Downloading flask-3.0.3-py3-none-any.whl (101 kB)
    101.7/101.7 kB 1.6 MB/s eta 0:00:00
Collecting importlib-metadata >= 3.6.0
  Downloading importlib_metadata-7.1.0-py3-none-any.whl (24 kB)
Collecting click >= 8.1.3
  Downloading click-8.1.7-py3-none-any.whl (97 kB)
    97.9/97.9 kB 3.1 MB/s eta 0:00:00
```

Servicio de Usuarios:

cd users-service

sudo docker build -t users-service .

```
(kali㉿kali)-[~/proyecto-final/users-service]
$ sudo docker build -t users-service .
Sending build context to Docker daemon 4.608kB
Step 1/7 : FROM python:3.9-slim
=> 4602238ffbd5
Step 2/7 : WORKDIR /app
=> Using cache
=> 61fa057720a0
Step 3/7 : COPY requirements.txt .
=> a3cda7f1cf96
Step 4/7 : RUN pip3 install --no-cache-dir -r requirements.txt
=> Running in 5e5bc68c2594
Collecting flask
  Downloading flask-3.0.3-py3-none-any.whl (101 kB)
    101.7/101.7 kB 632.3 kB/s eta 0:00:00
Collecting blinker>=1.6.2
  Downloading blinker-1.8.2-py3-none-any.whl (9.5 kB)
Collecting Werkzeug>=3.0.0
  Downloading werkzeug-3.0.3-py3-none-any.whl (227 kB)
    227.3/227.3 kB 714.0 kB/s eta 0:00:00
Collecting itsdangerous>=2.1.2
  Downloading itsdangerous-2.2.0-py3-none-any.whl (16 kB)
Collecting Jinja2>=3.1.2
  Downloading jinja2-3.1.4-py3-none-any.whl (133 kB)
    133.3/133.3 kB 4.2 MB/s eta 0:00:00
Collecting click>=8.1.3
  Downloading click-8.1.7-py3-none-any.whl (97 kB)
    97.9/97.9 kB 4.4 MB/s eta 0:00:00
Collecting importlib-metadata>=3.6.0
  Downloading importlib_metadata-7.1.0-py3-none-any.whl (24 kB)
Collecting zipp>=0.5
  Downloading zipp-3.19.2-py3-none-any.whl (9.0 kB)
Collecting MarkupSafe>=2.0
  Downloading MarkupSafe-2.1.5-cp39-cp39-manylinux_2_17_x86_64.manylinux2014_x86_64.whl (25 kB)
Installing collected packages: zipp, MarkupSafe, itsdangerous, click, blinker, Werkzeug, Jinja2, importlib-metadata, flask
Successfully installed Jinja2-3.1.4 MarkupSafe-2.1.5 Werkzeug-3.0.3 blinker-1.8.2 click-8.1.7 flask-3.0.3 importlib-metadata-7.1.0 itsdangerous-2.2.0 zipp-3.19.2
WARNING: Running pip as the 'root' user can result in broken permissions and conflicting behaviour with the system package manager. It is recommended to use a virtual environme
```

Servicio de Productos:

cd products-service

sudo docker build -t products-service .

```
(kali㉿kali)-[~/proyecto-final/products-service]
$ ls
app.py  Dockerfile  requirements.txt

(kali㉿kali)-[~/proyecto-final/products-service]
$ sudo docker build -t products-service .
Sending build context to Docker daemon 4.608kB
Step 1/7 : FROM python:3.9-slim
=> 4602238ffbd5
Step 2/7 : WORKDIR /app
=> Using cache
=> 61fa057720a0
Step 3/7 : COPY requirements.txt .
=> Using cache
=> a3cda7f1cf96
Step 4/7 : RUN pip3 install --no-cache-dir -r requirements.txt
=> Using cache
=> 26fe242eb3e7
Step 5/7 : COPY . .
=> 519da6e5dd99
Step 6/7 : EXPOSE 5002
=> Running in f3f4e4500570
Removing intermediate container f3f4e4500570
=> b550a5a56567
Step 7/7 : CMD ["python3", "app.py"]
=> Running in 229c34a3cc88
Removing intermediate container 229c34a3cc88
=> f3aca71b6e95
Successfully built f3aca71b6e95
Successfully tagged products-service:latest
```


Servicio de Pedidos:

cd orders-service

sudo docker build -t orders-service .

```
(kali㉿kali)-[~/proyecto-final/orders-service]
└─$ sudo docker build -t orders-service .
Sending build context to Docker daemon 4.608kB
Step 1/7 : FROM python:3.9-slim
─> 4602238ffbbc
Step 2/7 : WORKDIR /app
─> Using cache
─> 61fa057720a0
Step 3/7 : COPY requirements.txt .
─> Using cache
─> a3cda7f1cf96
Step 4/7 : RUN pip3 install --no-cache-dir -r requirements.txt
─> Using cache
─> 26fe242eb3e7
Step 5/7 : COPY . .
─> 19a429bfb891
Step 6/7 : EXPOSE 5004
─> Running in 44de2e2f4b5f
Removing intermediate container 44de2e2f4b5f
─> e9f5b8b2bbb1
Step 7/7 : CMD ["python3", "app.py"]
─> Running in 242985e0f950
Removing intermediate container 242985e0f950
─> 07e65ee8e318
Successfully built 07e65ee8e318
Successfully tagged orders-service:latest
```

Después de ejecutar estos comandos, deberías ver las imágenes Docker creadas en tu sistema:

sudo docker images

```
(kali㉿kali)-[~/proyecto-final]
└─$ ls
auth-service  orders-service  products-service  users-service

(kali㉿kali)-[~/proyecto-final]
└─$ sudo docker images
REPOSITORY          TAG             IMAGE ID        CREATED         SIZE
orders-service      latest         07e65ee8e318   5 minutes ago  137MB
products-service    latest        f3aca71b6e95   8 minutes ago  137MB
users-service       latest        705c32e5f3bd   11 minutes ago 137MB
auth-service        latest        a215d8f2e6fb   16 hours ago  137MB
python              3.9-slim      4602238ffbbc   2 months ago  126MB
```

Ahora que tenemos las imágenes Docker construidas, podemos proceder a ejecutar los contenedores de cada microservicio.

Ejecutar los contenedores Docker:

Abre una nueva terminal y ejecuta los contenedores de cada microservicio:

sudo docker run -p 5000:5000 auth-service

```

(kali㉿kali)-[~/proyecto-final/auth-service]
$ sudo docker run -p 5000:5000 auth-service
* Serving Flask app 'app'
* Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a p
roduction WSGI server instead.
* Running on all addresses (0.0.0.0)
* Running on http://127.0.0.1:5000
* Running on http://172.17.0.2:5000
Press CTRL+C to quit
* Restarting with stat
* Debugger is active!
* Debugger PIN: 367-699-043
172.17.0.1 - - [12/Jun/2024 09:10:28] "POST /login HTTP/1.1" 200 -
172.17.0.1 - - [12/Jun/2024 09:12:18] "GET /login HTTP/1.1" 405 -

```

Este comando ejecutará el contenedor del servicio de autenticación y mapeará el puerto 5000 del contenedor al puerto 5000 de mi máquina host. De esta manera, podre acceder al servicio de autenticación desde mi máquina.

sudo docker run -p 5001:5001 users-service

```

(kali㉿kali)-[~/proyecto-final]
$ sudo docker run -p 5001:5001 users-service
* Serving Flask app 'app'
* Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a
production WSGI server instead.
* Running on all addresses (0.0.0.0)
* Running on http://127.0.0.1:5001
* Running on http://172.17.0.2:5001
Press CTRL+C to quit
* Restarting with stat
* Debugger is active!
* Debugger PIN: 367-699-043
172.17.0.1 - - [13/Jun/2024 04:11:00] "GET /users HTTP/1.1" 200 -
172.17.0.1 - - [13/Jun/2024 04:14:40] "GET /users/1 HTTP/1.1" 200 -
172.17.0.1 - - [13/Jun/2024 04:15:03] "POST /users HTTP/1.1" 201 -

```

sudo docker run -p 5002:5002 products-service

```

(kali㉿kali)-[~/proyecto-final]
$ sudo docker run -p 5002:5002 products-service
[sudo] password for kali:
* Serving Flask app 'app'
* Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a
production WSGI server instead.
* Running on all addresses (0.0.0.0)
* Running on http://127.0.0.1:5002
* Running on http://172.17.0.3:5002
Press CTRL+C to quit
* Restarting with stat
* Debugger is active!
* Debugger PIN: 141-460-443
172.17.0.1 - - [13/Jun/2024 04:19:37] "GET /products HTTP/1.1" 200 -
172.17.0.1 - - [13/Jun/2024 04:20:03] "GET /products/1 HTTP/1.1" 200 -
172.17.0.1 - - [13/Jun/2024 04:20:17] "POST /products HTTP/1.1" 201 -

```

sudo docker run -p 5004:5004 orders-service

```
(kali@kali)-[~/proyecto-final]
$ sudo docker run -p 5004:5004 orders-service
* Serving Flask app 'app'
* Debug mode: on
WARNING: This is a development server. Do not use it in a production deployment. Use a
production WSGI server instead.
* Running on all addresses (0.0.0.0)
* Running on http://127.0.0.1:5004
* Running on http://172.17.0.5:5004
Press CTRL+C to quit
* Restarting with stat
* Debugger is active!
* Debugger PIN: 689-368-503
172.17.0.1 - - [13/Jun/2024 04:49:34] "GET /orders HTTP/1.1" 200 -
172.17.0.1 - - [13/Jun/2024 04:50:00] "GET /orders/1 HTTP/1.1" 200 -
172.17.0.1 - - [13/Jun/2024 04:50:14] "POST /orders HTTP/1.1" 201 -
```

Nota que estamos mapeando los puertos de los contenedores (5000) a puertos diferentes en la máquina host (5000, 5001, 5002, 5004) para evitar conflictos.

Probar los microservicios en contenedores Docker:

Abre una nueva terminal y prueba los microservicios utilizando los puertos mapeados:

Servicio de autenticación

curl -X POST -H "Content-Type: application/json" -d '{"username":"testuser","password":"testpassword"}' <http://localhost:5000/login>

```
(kali@kali)-[~/proyecto-final]
$ curl -X POST -H "Content-Type: application/json" -d '{"username":"testuser","password":"testpassword"}' http://localhost:5000/login
{"token": "abc123"}
```

Esto indica que el servicio de autenticación está funcionando correctamente dentro del contenedor Docker.

Servicio de usuarios

curl <http://localhost:5001/users>

```
(kali㉿kali)-[~/proyecto-final]
$ curl http://localhost:5001/users
[
  {
    "id": 1,
    "name": "John Doe"
  },
  {
    "id": 2,
    "name": "Jane Doe"
  }
]
```

curl <http://localhost:5001/users/1>

```
(kali㉿kali)-[~/proyecto-final]
$ curl http://localhost:5001/users/1
{
  "id": 1,
  "name": "John Doe"
}
```

curl -X POST -H "Content-Type: application/json" -d '{"name":"John Doe"}' http://localhost:5001/users

```
(kali㉿kali)-[~/proyecto-final]
$ curl -X POST -H "Content-Type: application/json" -d '{"name":"John Doe"}' http://localhost:5001/users
{
  "id": 3,
  "name": "John Doe"
}
```

Servicio de productos

curl http://localhost:5002/products

curl http://localhost:5002/products/1

curl -X POST -H "Content-Type: application/json" -d '{"name":"Producto 3","price":20.99}'
http://localhost:5002/products

```

(kali㉿kali)-[~/proyecto-final]
$ curl http://localhost:5002/products
[
  {
    "id": 1,
    "name": "Producto 1",
    "price": 10.99
  },
  {
    "id": 2,
    "name": "Producto 2",
    "price": 15.5
  }
]

(kali㉿kali)-[~/proyecto-final]
$ curl http://localhost:5002/products/1
{
  "id": 1,
  "name": "Producto 1",
  "price": 10.99
}

(kali㉿kali)-[~/proyecto-final]
$ curl -X POST -H "Content-Type: application/json" -d '{"name":"Producto 3","price":20.99}' http://localhost:5002/products
{
  "id": 3,
  "name": "Producto 3",
  "price": 20.99
}

```

Servicio de pedidos

curl http://localhost:5004/orders

curl http://localhost:5004/orders/1

curl -X POST -H "Content-Type: application/json" -d '{"user_id":1,"total":25.99}'
http://localhost:5004/orders

```

(kali㉿kali)-[~/proyecto-final]
$ curl http://localhost:5004/orders
[
  {
    "id": 1,
    "total": 25.99,
    "user_id": 1
  },
  {
    "id": 2,
    "total": 48.75,
    "user_id": 2
  }
]

(kali㉿kali)-[~/proyecto-final]
$ curl http://localhost:5004/orders/1
{
  "id": 1,
  "total": 25.99,
  "user_id": 1
}

(kali㉿kali)-[~/proyecto-final]
$ curl -X POST -H "Content-Type: application/json" -d '{"user_id":1,"total":25.99}' h
http://localhost:5004/orders
{
  "id": 3,
  "total": 25.99,
  "user_id": 1
}

```

