# Introducción

## Descripción general del proyecto

Este proyecto es un sistema de gestión de pedidos para un restaurante, implementado como una aplicación web robusta y escalable. Utiliza una arquitectura moderna y distribuida para manejar eficientemente los pedidos, verificar la disponibilidad de ingredientes y coordinar la preparación de alimentos.

## Objetivos

1. Desarrollar un sistema eficiente de gestión de pedidos para restaurantes.
2. Implementar una arquitectura escalable y distribuida.
3. Asegurar la disponibilidad de ingredientes en tiempo real.
4. Optimizar el proceso de preparación de alimentos.
5. Proporcionar una API robusta para la interacción con el sistema.
6. Garantizar la compatibilidad con IPv4 e IPv6.

# Arquitectura del sistema

El sistema se compone de varios componentes principales:

* **Aplicación Web**: Desarrollada con Flask, maneja las peticiones HTTP y las rutas de la aplicación.
* **Tareas Asíncronas**: Gestionadas por Celery, permite la ejecución de tareas en segundo plano.
* **Base de Datos y Almacenamiento**: Utiliza Redis tanto como broker de mensajes para Celery como backend de resultados.
* **Contenerización**: Docker se utiliza para contenerizar la aplicación, facilitando la implementación y escalabilidad.

## Tecnologías utilizadas (Flask, Celery, Redis, Docker)

**Flask**: Framework web para Python utilizado para desarrollar la aplicación web.

**Celery**: Biblioteca para la ejecución de tareas asíncronas.

**Redis**: Base de datos en memoria utilizada como broker de mensajes y backend de resultados.

**Docker**: Plataforma de contenerización que permite empaquetar la aplicación y sus dependencias.

# Componentes principales

## Aplicación web (Flask)

Desarrollado con Flask, un framework web de Python ligero y flexible. Proporciona endpoints RESTful para la gestión de menús, creación de pedidos y consulta de estados.

## Estructura de archivos

final/

- app/

-- \_\_init\_\_.py

-- celery\_app.py

-- clients.py

-- config.py

-- despensa.py

-- order.py

-- server.py

-- task.py

-- utils.py

- doc/

- task/

-- \_\_init\_\_.py

-- prepare.py

- run.py

- TODO.md

## Rutas y funcionalidades

**GET /menu**

* **Funcionalidad:** Retorna el menú disponible del restaurante.

**POST /pedido**

* **Funcionalidad:** Crea un nuevo pedido y inicia el proceso de verificación y preparación.

**GET /status/<task\_id>**

* **Funcionalidad:** Consulta el estado de un pedido específico.

## Tareas asíncronas (Celery)

Utiliza Celery para manejar tareas de larga duración de manera asíncrona, como la verificación de ingredientes y la preparación de pedidos. Esto permite que la aplicación sea más responsive y pueda manejar múltiples pedidos simultáneamente.

## Configuración

**celery\_app.py**: Configuración de Celery dentro de la aplicación.

**Configuración de broker y backend**: Uso de Redis como broker de mensajes y backend de resultados.

## Tareas implementadas

**check\_ingredients**:

* Verifica la disponibilidad de ingredientes para un pedido.
* Actualiza el estado del pedido según la disponibilidad.

**prepare**:

* Inicia la preparación de los productos en el pedido.
* Actualiza el estado de cada producto y del pedido general.

**Tareas específicas de preparación** (prepare\_hamburguesa, prepare\_pizza, etc.):

* Simulan la preparación de productos específicos.

## Base de datos y almacenamiento (Redis)

Redis se utiliza en dos roles cruciales: Almacenamiento y mensajería

## Uso como broker de mensajes

Para Celery, facilitando la comunicación entre la aplicación web y los workers de Celery.

## Uso como backend de resultados

Para almacenamiento de los estados de las tareas y permitiendo consultas rápidas sobre el progreso de los pedidos.

# Docker y containerización

Todo el sistema está en containers utilizando Docker, lo que facilita el despliegue, la escalabilidad y la consistencia entre diferentes entornos.

## Estructura del docker-compose.yml

version: '3'

services:

web:

build:

context: .

dockerfile: Dockerfile

ports:

- "8080:8080"

environment:

- FLASK\_APP=run.py

depends\_on:

- redis

volumes:

- .:/app

celery:

build:

context: .

dockerfile: Dockerfile.celery

depends\_on:

- redis

environment:

- CELERY\_BROKER\_URL=redis://redis:6379/0

- CELERY\_RESULT\_BACKEND=redis://redis:6379/0

volumes:

- .:/app

redis:

image: "redis:alpine"

## Beneficios de la contenerización en este proyecto

**Aislamiento**: Cada componente del sistema se ejecuta en su propio contenedor.

**Portabilidad**: Los contenedores pueden ejecutarse en cualquier entorno compatible con Docker.

**Escalabilidad**: Facilita el escalado horizontal de los servicios.

## Servidor de producción

Gunicorn actúa como servidor WSGI, proporcionando un entorno robusto y eficiente para ejecutar la aplicación Flask en producción.

**Web Server Gateway Interface**. Es una especificación que describe cómo se comunica **un servidor web** con **una aplicación web**, y cómo se pueden llegar a encadenar diferentes aplicaciones web para procesar una solicitud/petición (o *request*).

# Flujo de trabajo de la aplicación

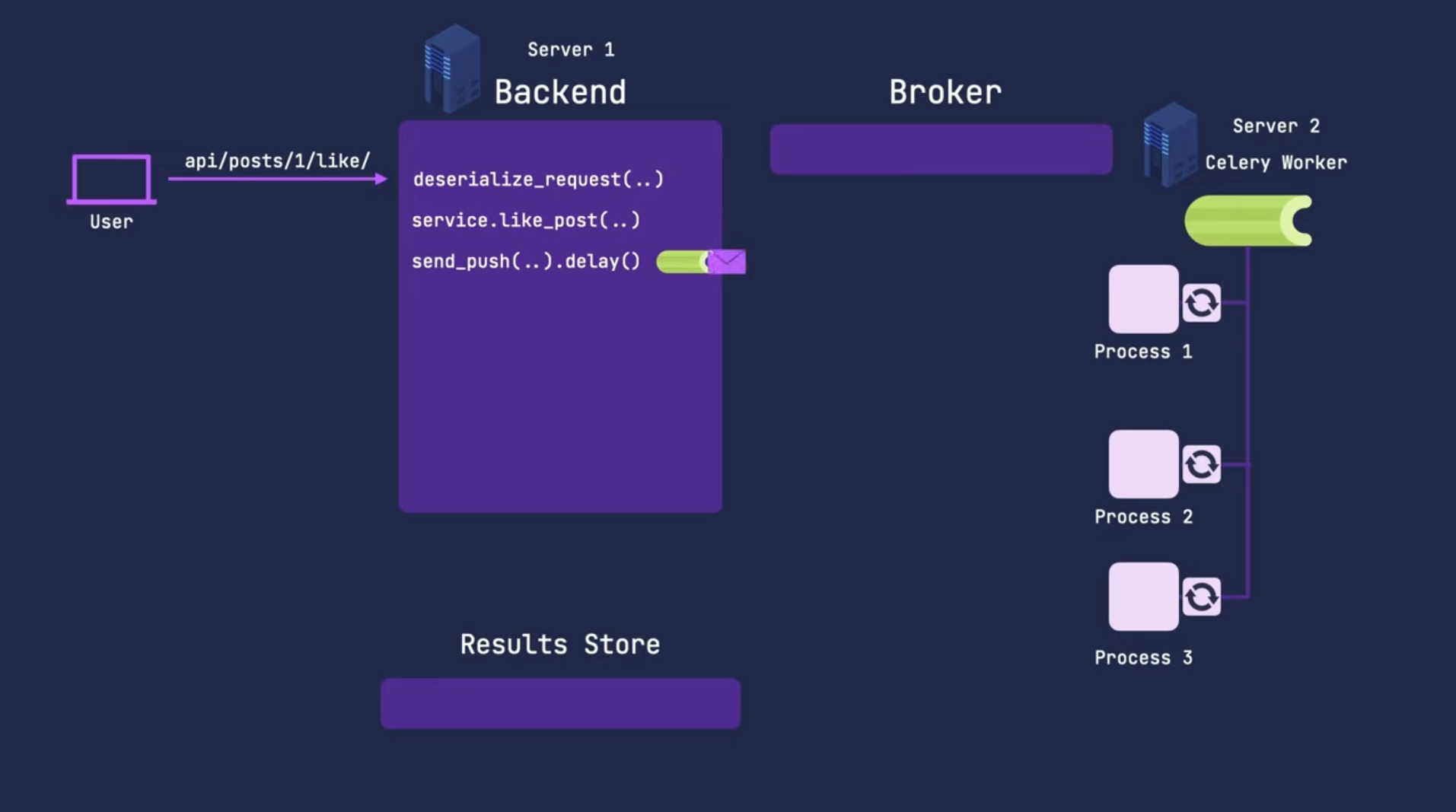
1. Los clientes pueden consultar el menú y realizar pedidos a través de la API.
2. Cada pedido desencadena una serie de tareas asíncronas para verificar ingredientes y preparar los alimentos.
3. El estado de cada pedido se puede consultar en tiempo real.

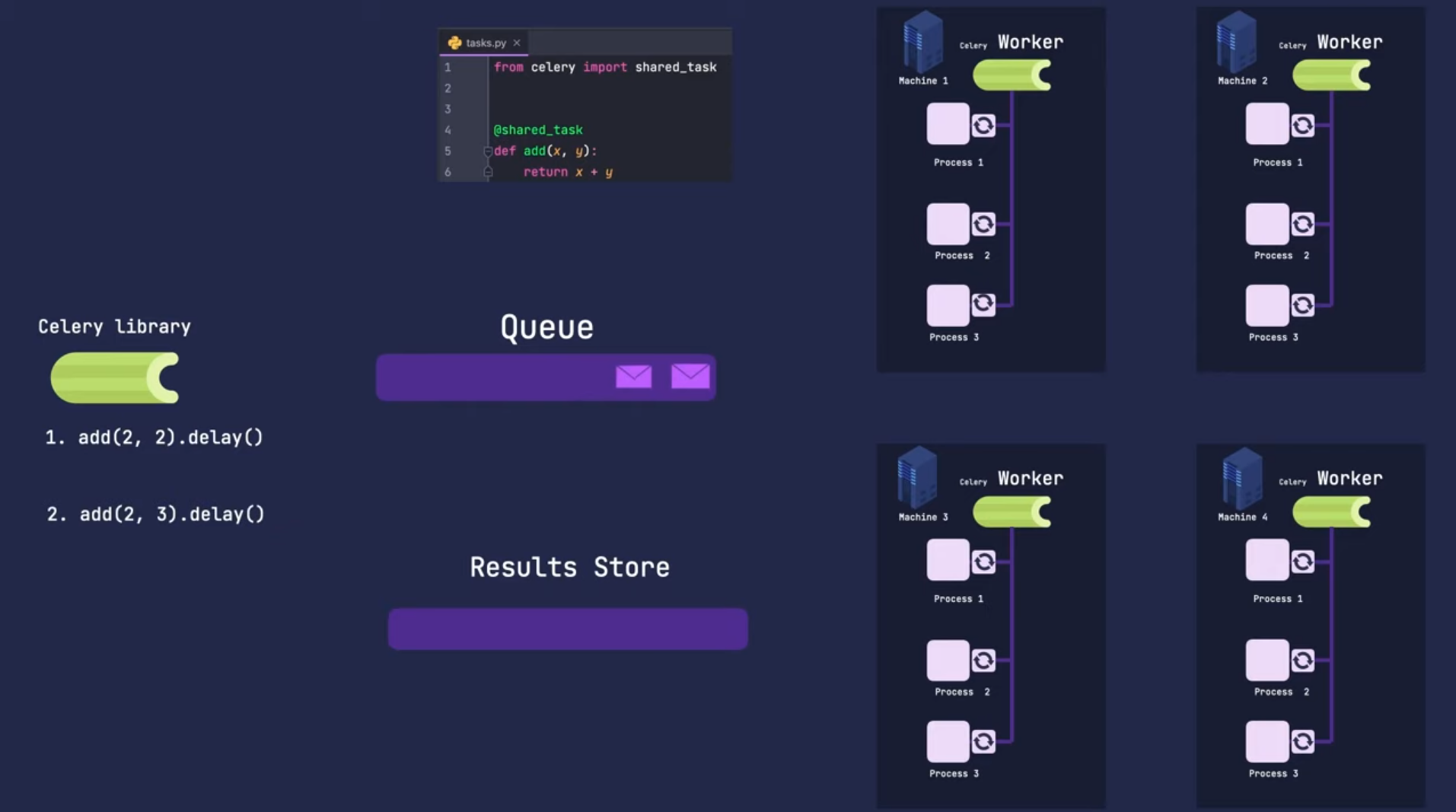
## Explicación paso a paso

1. **Solicitud HTTP**: El usuario realiza una solicitud a la aplicación web Flask.
2. **Procesamiento de la solicitud**:

* Cuando llega una solicitud HTTP, Gunicorn la recibe.
* Gunicorn pasa la solicitud a uno de sus workers.
* El worker, que tiene cargada tu aplicación Flask, procesa la solicitud usando las rutas y lógica definidas..
* Flask maneja la solicitud y, si es necesario, delega tareas a Celery.

1. **Ejecución de tareas**: Celery ejecuta las tareas asíncronas y almacena los resultados en Redis.
2. **Respuesta HTTP**: Flask devuelve la respuesta al usuario





# Manejo de IPv4 e IPv6

La aplicación está configurada para funcionar tanto con IPv4 como con IPv6, asegurando compatibilidad con diferentes configuraciones de red.

## Configuración

**Configuración de Docker**: Configuración de Docker para soportar IPv6.

yaml:

services:

web:

...

networks:

- ipv6\_network

networks:

ipv6\_network:

enable\_ipv6: true

Dockerfile & Dockerfile.celery:

EXPOSE 8080

Para habilitar IPv6 en Docker, puedes editar el archivo /etc/docker/daemon.json (créalo si no existe) y añade:

json:

{ "ipv6": true, "fixed-cidr-v6": "2001:db8:1::/64" }

Luego reinicia el servicio Docker:

sudo systemctl restart docker

## Explicación del mapeo IPv6 a IPv4

Todo esto permite que Docker haga un mapeo de IPv6 a IPv4 cuando reciba in IPv4 en la direccion de IPv6.

Para asegurarte de que IPv4-mapped IPv6 addresses están habilitadas:

sysctl net.ipv6.bindv6only

Si el resultado es 0, significa que las conexiones IPv4 se mapean automáticamente.

# Pruebas y depuración

## Métodos para probar la aplicación (Postman, curl)

**Postman**: Uso de Postman para enviar solicitudes HTTP y verificar las respuestas.

**curl**: Uso de curl para probar endpoints desde la línea de comandos.

# Links de interés

* Configuracion de docker para IPv6 e IPv4: https://docs.docker.com/config/daemon/ipv6/
* Documentacion de Celery: <https://docs.celeryq.dev/en/stable/>
* Video animado de Celery: [Python. Celery concepts animated](https://youtu.be/TzVkED3y3Ig?si=hKLaWAJf9LZJWtrq)