

PDF de explicacion paso a paso

La finalidad de esta explicación es proporcionar una visión más clara sobre cómo podemos mejorar el rendimiento de nuestros sistemas desarrollados en **Django** de manera sincrónica, sin agregar complejidad innecesaria. También quiero mostrar las alternativas disponibles para resolver estos desafíos de manera eficiente.

Tecnologías a utilizar

Para implementar las tareas en segundo plano y mejorar la eficiencia de nuestro sistema, usaremos las siguientes tecnologías. Asegúrate de tener **Python** instalado en tu sistema antes de continuar. Las dependencias estarán listadas en un archivo `requirements.txt` para facilitar su instalación:

- **Django**
- **Django REST Framework**
- **Celery**

Antes de iniciar el proyecto debemos de crear un entorno virtual para aislar las dependencias de nuestro proyecto para evitar conflictos, por lo que ejecutaremos el siguiente comando:

```
python -m venv nombre-entorno
```

Una vez creado el entorno accederemos a él por la línea de comandos, para poder activar el mismo, dependiendo del sistema operativo esto va a cambiar realmente:

Windows:

```
cd ruta/venv/Scripts && activate
```

Linux:

```
source ruta/venv/bin/activate
```

Procuremos que dentro del **cmd** el entorno este habilitado como se muestra aqui:

```
(venv) C:\Users\luisd\OneDrive\Escritorio\django-rabbit>
```

Ya teniendo estos pasos completos, podemos proceder a instalar las dependencias.

Comando de instalación de dependencias y configuraciones

```
pip install django djangorestframework celery asyncio
```

Ya una vez hayan sido instaladas procederemos a crear un proyecto de django:

```
// Creamos un proyecto y colocamos . al final para que no nos
django-admin startproject django-rabbit .
// Creamos una app de usuarios directamente
django-admin startapp users
```

Agregamos la app creada de users y los modulos de celery y django-rest para poder trabajar con ellos y la configuracion de nuestro servidor smtp para enviar correos:

```
#settings.py

INSTALLED_APPS = [
    'rest_framework',
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
```

```

        'users',
        'celery'
    ]

    #Configuracion de email, en este caso se utiliza gmail (la co
    EMAIL_BACKEND = 'django.core.mail.backends.smtp.EmailBackend'
    EMAIL_HOST = 'smtp.gmail.com'
    EMAIL_PORT = 587
    EMAIL_USE_TLS = True
    EMAIL_HOST_USER = os.environ.get("EMAIL_HOST_USER")
    EMAIL_HOST_PASSWORD = os.environ.get("EMAIL_HOST_PASSWORD")

```

Iniciamos el servidor ejecutando el comando en la terminal, así activando el proyecto de django:

```
python manage.py runserver
```

Dentro de nuestro proyecto de django creado, agregaremos un archivo llamado celery.py, con este definiremos e instanciaremos los ajustes necesarios que se requieren para poder trabajar con celery, como los siguientes:

```

#celery.py

from __future__ import absolute_import, unicode_literals
import os
from celery import Celery

# Establecer el entorno de Django y le pasamos la configuraci
#mi proyecto en este caso (core)
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'core.setting

#Creamos una instancia de nuestro proyecto en celery
app = Celery('core')

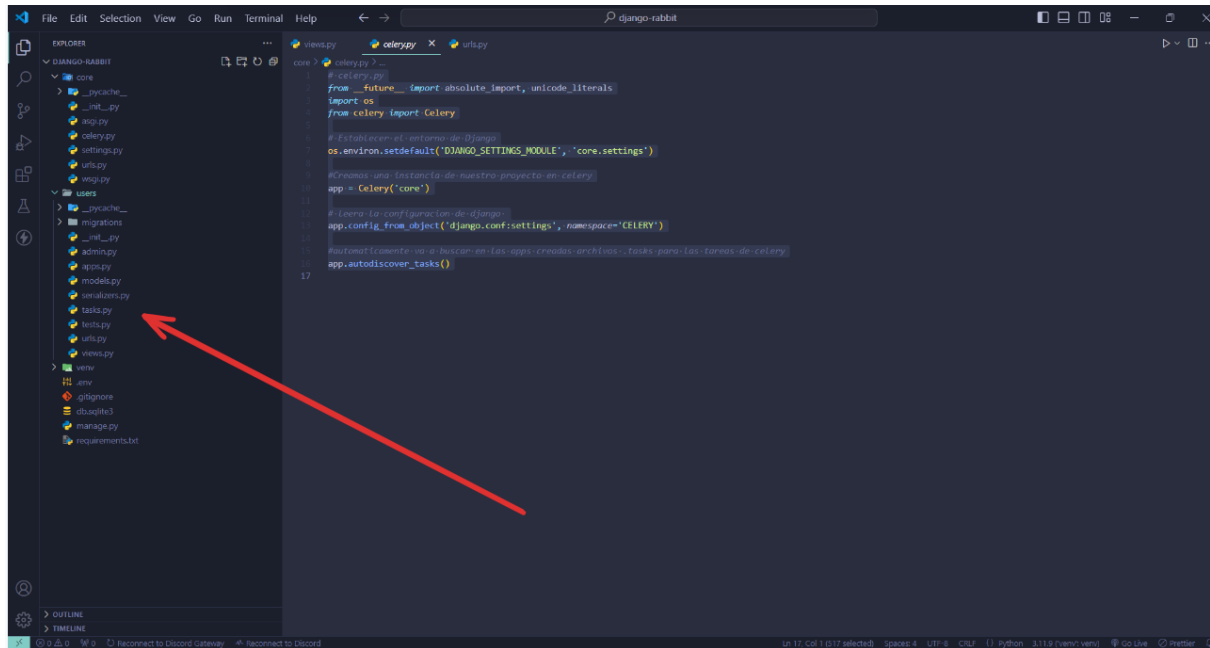
# leera la configuracion de django
app.config_from_object('django.conf:settings', namespace='CEL

```

```
#automaticamente va a buscar en las apps creadas archivos .ta  
app.autodiscover_tasks()
```

Configuración de archivo tasks.py en nuestra app

En mi caso la app en la que utilizare estos shared tasks será en el de usuarios



Las tareas van a depender de lo que requieras hacer, lo recomendable es que las utilices para tareas que requieran de mucho tiempo y no se bloquee el hilo principal por el que corre nuestro proyecto de django.



Nota: Celery se encargara de mapear la tareas automáticamente por nosotros, buscando así todos los archivos de nuestras apps que tengan un tasks.py

```
#tasks.py  
from celery import shared_task  
from core import settings  
from django.core.mail import send_mail  
import core
```

```

#Tarea de celery que se ejecutara en segundo plano luego de q
@shared_task
def send_email_task( message):
    #Lista de recipientes a la que llegara el correo
    recipient_list = [core.settings.EMAIL_HOST_USER]
    #subject del correo
    subject = "Correo desde celery utilizando el broker de Ra
    #desde donde se enviara el correo
    email_from = settings.EMAIL_HOST_USER
    #funcion de django que nos permite enviar el correo
    send_mail(subject, message, email_from, recipient_list)

```

Ya teniendo esto crearemos nuestra urls para que estas den respuesta mediante una vista relacionadas a ellas y podamos crear usuarios dentro del sistema siendo validados por nuestro serializador relacionado a nuestro modelo de users (**views.py, urls.py, serializers.py**):

```

#views.py
from rest_framework import status
from rest_framework.response import Response
from rest_framework.viewsets import ModelViewSet
from rest_framework.views import APIView
from django.contrib.auth.models import User
from .serializers import UserSerializer
import core.settings
from django.core.mail import send_mail
from dotenv import load_dotenv
import time
from .tasks import send_email_task
load_dotenv()

class UserViewSet(ModelViewSet):

    serializer_class = UserSerializer
    queryset = User.objects.all()

```

```

#funcion que handlea la creacion de un nuevo usuario
def create(self, request, *args, **kwargs):

    # se obtiene el serializador para instanciar la data
    serializer = self.get_serializer(data=request.data)
    #Se ejecuta la validacion del serializador y lanzamos
    serializer.is_valid(raise_exception=True)

    # al ser valido llegara aqui por lo que se guarda el
    self.perform_create(serializer)

    message = f'Se ha creado un nuevo Usuario dentro del

    #se ejecuta la funcion que celery alojara en la cola
    send_email_task.delay(message)

    #Se devuelve una respuesta con la data repres
    return Response(serializer.data, status=status.HTTP_2

```

Aqui tendremos la configuracion de las urls y la definición del serializador para el modelo de usuario.

```

#urls.py
from django.urls import path, include
from .views import *
from rest_framework.routers import DefaultRouter

#Registramos el router para agrear vistas basadas en ViewSets
router = DefaultRouter()
router.register("user",UserViewSet, basename="user")

urlpatterns = [
    #incluimos las urls registradas en el router
    path("",include(router.urls))

]

```

```
#serializers.py
from rest_framework import serializers
from django.contrib.auth.models import User

class UserSerializer(serializers.ModelSerializer):
    class Meta:
        model = User
        fields = ("first_name", "last_name", "username", "password")
```

Realizamos las migraciones necesarias para que django cree las tablas que requiere para poder trabajar

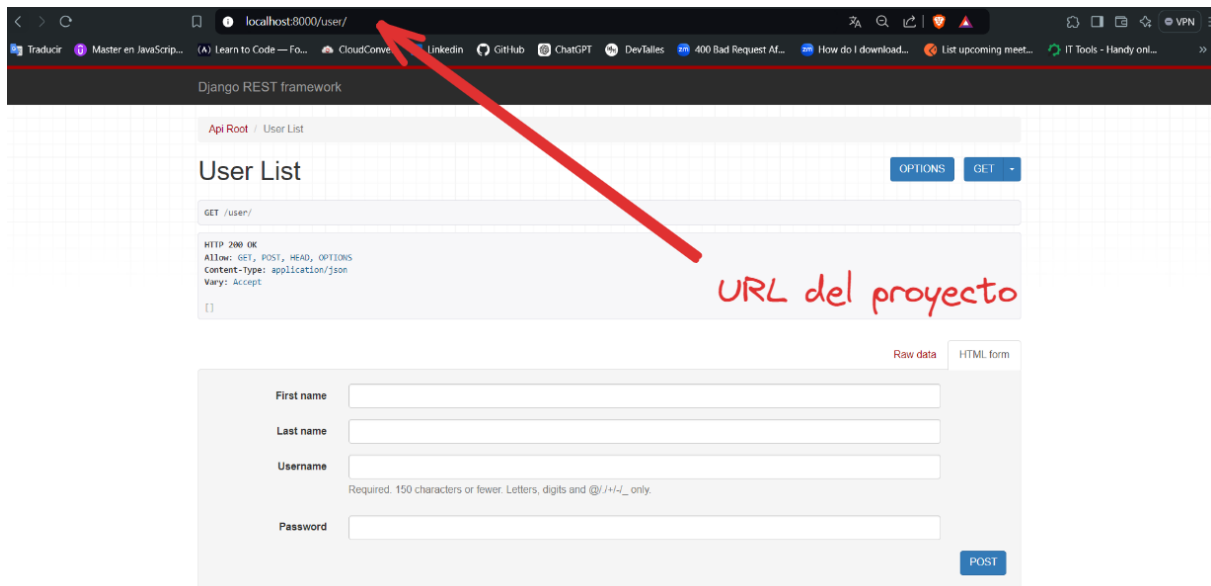
```
#primero realizamos las migraciones
python manage.py makemigrations
#aplicamos las migraciones
python manage.py migrate
```

Teniendo en cuenta que mi app de usuarios es una app externa y no del proyecto perced, debemos de integrar las urls desarrolladas en el core del proyecto:

```
from django.contrib import admin
from django.urls import path, include

urlpatterns = [
    path('admin/', admin.site.urls),
    #aqui estamos registrando las urls de la app de users
    path("", include("users.urls"))
]
```

Ya teniendo todas estas acciones deberiamos de poder usar nuestra aplicacion correctamente haciendo solicitudes al endpoint generado que seria **http:localhost:8000/user:**



Y el servidor corriendo sin problemas:

```
System check identified no issues (0 silenced).
September 16, 2024 - 21:40:23
Django version 5.1.1, using settings 'core.settings'
Starting development server at http://127.0.0.1:8000/
Quit the server with CTRL-BREAK.

[16/Sep/2024 21:40:23] "GET /user/ HTTP/1.1" 200 9370
```

Hasta el momento solo nos faltarían 2 pasos esenciales que es ejecutar celery y configurar nuestro servidor de rabbitMQ para que funcione de broker para celery.

Instalación de servidor de rabbitMQ con docker:

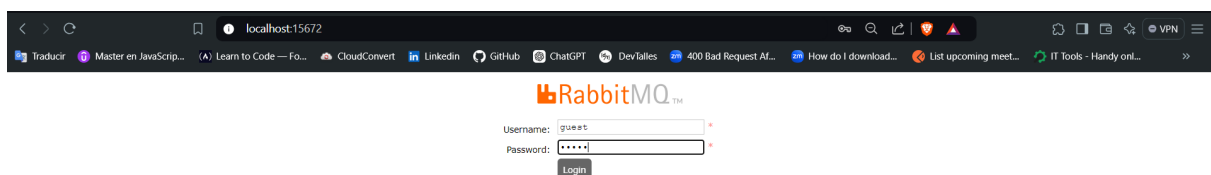
Con este comando estaremos creando un contenedor llamado rabbitmq que va a correr en el puerto 5672 de nuestra maquina y 5672 del contenedor, se utilizara la imagen de rabbitmq con los tags de 3.13-management. Y correrán su interfaz Web por medio del puerto 15672 de nuestra maquina y del contenedor.

```
docker run -d --rm --name rabbitmq -p 5672:5672 -p 15672:15672
```

Así podremos ver como se vio la ejecución del comando:


```
C:\Users\luisd>docker run -d --rm --name rabbitmq -p 5672:5672 -p 15672:15672 rabbitmq:3.13-management
Unable to find image 'rabbitmq:3.13-management' locally
3.13-management: Pulling from library/rabbitmq
857cc8cb19c0: Pull complete
5e224540077f: Pull complete
710bde4cd4dd: Pull complete
0600d9b5a93a: Pull complete
84165a2781c0: Pull complete
a0ab77f2d6f1: Pull complete
bd432105f7d1: Pull complete
9fb15500b04f: Pull complete
6fa2cd0de6e2: Pull complete
e408a9d2500f: Pull complete
Digest: sha256:6d545da940de0217b72b2957efbee589b383771ddaf117be30830fd3d9b198a1
Status: Downloaded newer image for rabbitmq:3.13-management
3ae329e17c9fe746fda3055b39c7fe033191a1019c2aa78980e23f86dd29d2f9
```

Una vez realizado con éxito podremos acceder a la interfaz web que rabbit genera para la gestión de nuestros queues y conexiones accediendo a <http://localhost:15672> y deberíamos de poder ver la siguiente interfaz:



Por defecto, RabbitMQ crea un **usuario**: `guest` y una **contraseña**: `guest`, lo cual se recomienda cambiar en un entorno de producción.

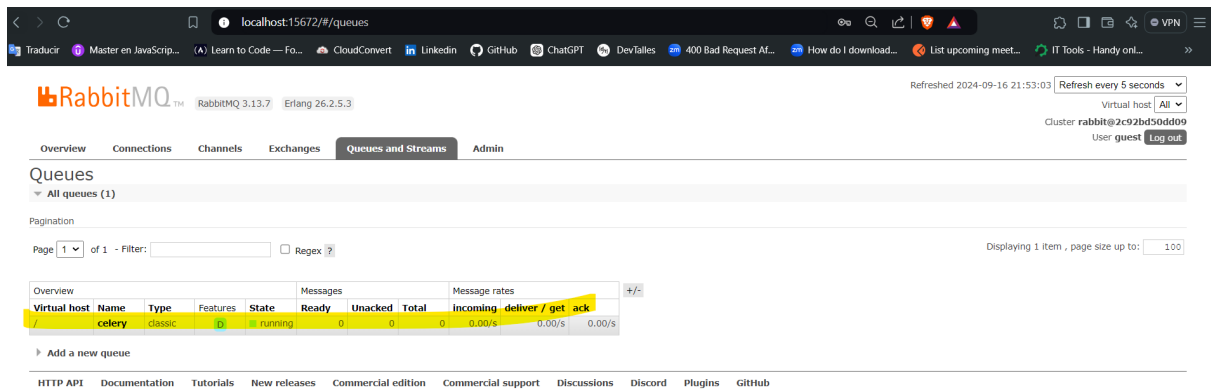
Una vez RabbitMQ esté configurado, procedemos con Celery:

En Windows, se recomienda ejecutar el siguiente comando en la terminal para evitar algunos fallos. En Linux no se necesitan los parámetros `concurrency` ni `-P`.

Es necesario ejecutar Celery para crear un worker que procesará las tareas en la cola por defecto, llamada **celery**. Si celery no es configurado las tareas no estarán escuchando a ningún queue por lo que no se estarían aislando del

proceso central de django. RabbitMQ enviará los mensajes a Celery, y usando `loglevel info`, podremos ver más detalles útiles para la depuración.

```
celery -A core worker -Q celery --loglevel=INFO --concurrency
```



Virtual host	Name	Type	Features	State	Messages			Message rates		
					Ready	Unacked	Total	Incoming	deliver	get
/	celery	classic		running	0	0	0	0.00/s	0.00/s	0.00/s

Queue de rabbit:

Ejecucion del comando de Celery:

```
(venv) C:\Users\luisd\OneDrive\Escritorio\django-rabbit> celery -A core worker -Q celery --loglevel=INFO --concurrency 1 -P solo
[2024-09-16 21:54:18,217: WARNING/MainProcess] No hostname was supplied. Reverting to default 'localhost'

----- celery@DESKTOP-72T1KEQ v5.4.0 (opal) -----
*****
Windows-10-10.0.22631-SP0 2024-09-16 21:54:18
***
[config]
> app: core:0x128d5747850
> transport: amqp://guest:**@localhost:5672//
> results: disabled://
> concurrency: 1 (solo)
> task events: OFF (enable -E to monitor tasks in this worker)
*****
[queues]
> celery exchange=celery(direct) key=celery

[tasks]
. users.tasks.send_email_task

[2024-09-16 21:54:18,305: WARNING/MainProcess] C:\Users\luisd\OneDrive\Escritorio\django-rabbit\venv\Lib\site-packages\celery\worker\consumer\consumer.py:508: CPendingDeprecationWarning: The broker_connection_retr
try configuration setting will no longer determine
whether broker connection retries are made during startup in Celery 6.0 and above.
If you wish to retain the existing behavior for retrying connections on startup,
you should set broker_connection_retry_on_startup to True.
warnings.warn(

[2024-09-16 21:54:18,336: INFO/MainProcess] Connected to amqp://guest:**@127.0.0.1:5672//
[2024-09-16 21:54:18,339: WARNING/MainProcess] C:\Users\luisd\OneDrive\Escritorio\django-rabbit\venv\Lib\site-packages\celery\worker\consumer\consumer.py:508: CPendingDeprecationWarning: The broker_connection_re
try configuration setting will no longer determine
whether broker connection retries are made during startup in Celery 6.0 and above.
If you wish to retain the existing behavior for retrying connections on startup,
you should set broker_connection_retry_on_startup to True.
warnings.warn(

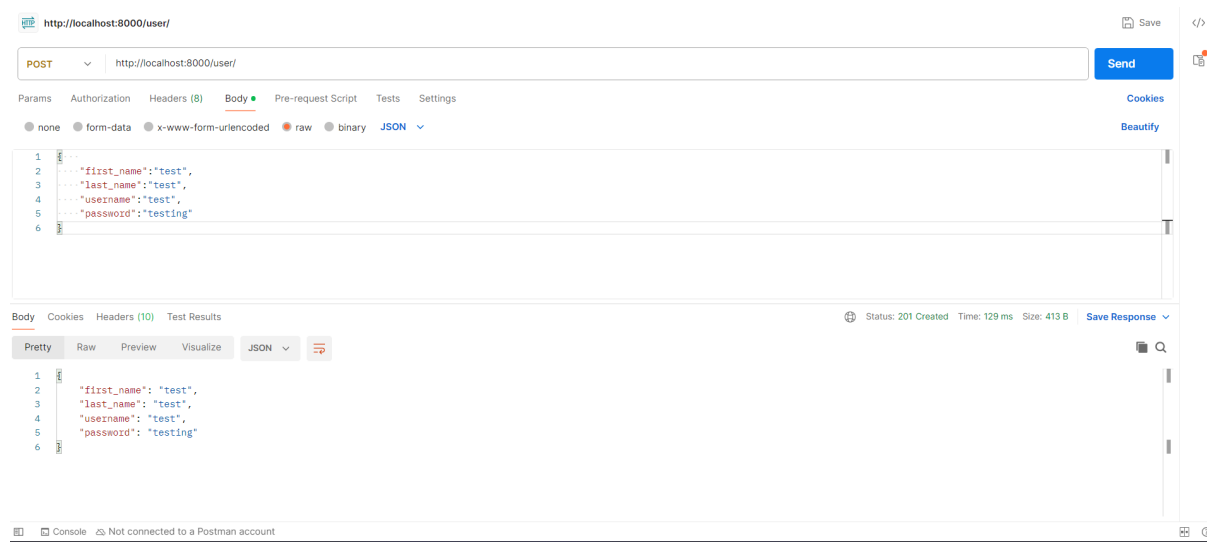
[2024-09-16 21:54:18,373: INFO/MainProcess] mingle: searching for neighbors
[2024-09-16 21:54:18,428: WARNING/MainProcess] No hostname was supplied. Reverting to default 'localhost'
[2024-09-16 21:54:19,492: INFO/MainProcess] mingle: all alone
[2024-09-16 21:54:19,594: INFO/MainProcess] celery@DESKTOP-72T1KEQ ready.
.
```

Cosas a destacar:

Debemos de tener el entorno virtual activado de nuestro proyecto de django en el que tenemos celery configurado, pasandole como parametro el mismo para correr el comando.

Celery tiene una configuracion por defecto que hara la conexion con rabbit, pero si se cambiaron las credenciales de rabbit con otro usuario se debera de modificarlas nuevamente para que la conexion funcione correctamente.

Ya teniendo en cuenta esto simplemente haremos una solicitud con **postman** a nuestro endpoint para ver el output que esto nos genera:



Correo recibido:

📧 ☆ yo

Correo desde celery utilizando el broker de RabbitMQ - Se ha creado un nuevo Usuario dentro del sistema de prueba

📅 🗑️ 📧 ⌚