# DOCUMENTAZIONE FUNZIONALE – PRE PROCESSING.

## Introduzione:

5



Custom Pre-processing service

GR Python - API

3

2

1

4

Pre-Processing #

1. Cliente Invia file a Greenrouter
2. Greenrouter: rileva nuovo file, seleziona il preprocesso corrispondente e manda il file a API
3. API: schedula il pre-processing.
4. process\_retriever: Gestisce la coda dei pre-processi (max di 4 elaborazioni parallele), e lancia pre\_processor\_launcher
5. pre\_processor\_launcher:
   1. seleziona e avvia il preprocessing che trasforma il file in un formato “GR-friendly”.
   2. Riceve il file elaborato e lo rimanda a Greenrouter per il calcolo della CO2e

## Codice Python – pre\_processing\_api\_v1

### Introduzione:

Il servizio permette la ricezione dei file da elaborare da GR.

### Logica Upload\_Files:

|  |  |  |
| --- | --- | --- |
| TABLE NAME | COLUMN\_NAME | Descrizione |
| Import | Id | Codice Identificativo import |
| Import | gr\_codice | (Input) Codice identificativo della sotto istanza (azienda del cliente) |
| Import | gr\_auth\_version | (input) versione dell’import:   * v1 * v2 * v3 |
| Import | gr\_response url | (input) Indirizzo dell’istanza a cui rispondere. |
| Import | nome\_file | (input) PATH/FILE |
| Import | Status | Auto  0 = da elaborare (default, dopo l’import) |

#### Costruttore (\_\_init\_\_)

Il costruttore della classe Upload\_Files definisce una lista che conterrà i nomi dei file e Il tipo di estensione permesso (in questo caso è permesso soltanto l’estensione di tipo “.csv”).

#### Metodi:

##### Allowed\_files:

Il seguente metodo controlla se il file da caricare contiene l’estensione di tipo “.csv”

##### Post:

Il seguente metodo è quello principale, indica quali sono i parametri chiavi da popolare durante la chiamata:

* “files[]”
* “process”
* “codice”
* “out\_version”
* “response\_url”

Fatta la chiamata, viene controllata se i parametri chiavi siano completi e popolati, verrà poi estratto il file allegato nel parametro “files[]” e poi controllato se è contenuto tra i file permessi (“Allowed\_Files”). Se il file è permesso viene creato un nuovo nome per il file aggiungendo all’originale la data e l’ora in cui è stato caricato, questo è per evitare che sovrascriva un file dello stesso nome caricato in precedente, infine il file caricato sarà salvato in una cartella situata nella stessa directory in cui si trova il progetto mentre la chiamata è salvata nella tabella “import” situata nel database “pre\_processing”.

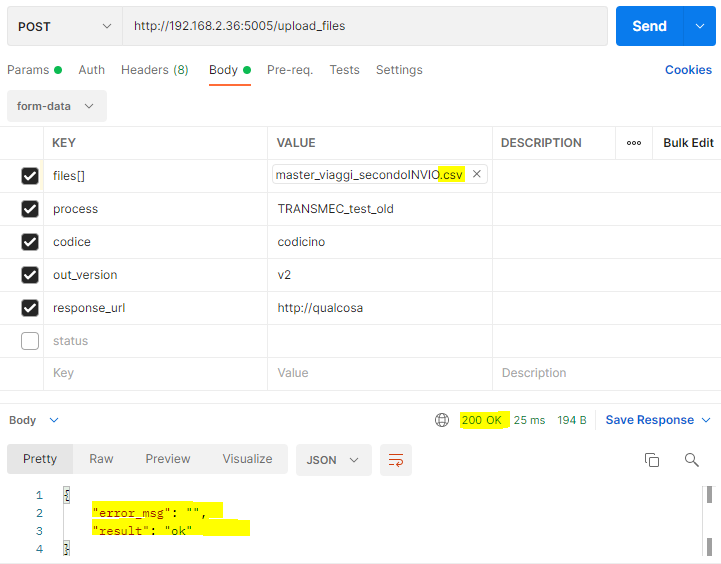
#### OUTPUT:

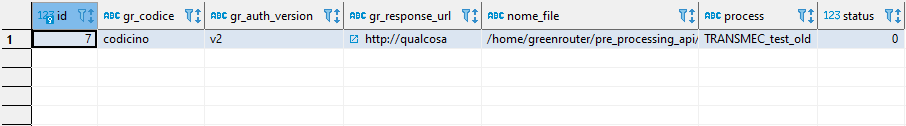
L’output è gestito dallo status code che identifica il codice d’errore o successo e da un dizionario che contiene il risultato e messaggio d’errore:  
{‘**result**’: ‘ ‘ , ‘**error\_msg’**: ’ ’} alla fine della chiamata viene trasformato in formato json.  
Esistono 3 casi di output:

1. **result**: ‘ok’ e **error\_msg**: ‘ ‘ 🡪 Import Completato senza errori.  
   **status\_code**: 200 🡪 Success
2. **result**: ‘err’ e **error\_msg**: ‘|Missing parameters in request|’ 🡪 Errore sui parametri   
   **status\_code**: 401 🡪 Error/UNAUTHORIZED
3. **result**: ‘err’ e **error\_msg**: ‘|nomefile.example||Some files were not uploaded please retry|’ 🡪 File da caricare in allegato non è permesso (estensione file diverso da .csv).  
   **status\_code**: 401 🡪 Error.

##### Esempi:

###### Caso 1

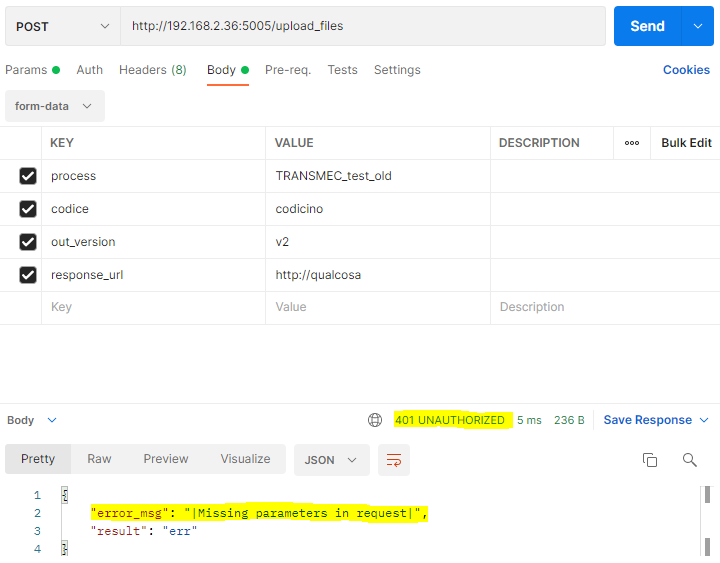
In questo caso la chiamata è completa con i parametri necessari e il file in allegato ha l’estensione corretta.  


Chiamata salvata nella tabella ‘import’ dal ‘pre\_processing’ database.  


File salvato nella cartella definita.  

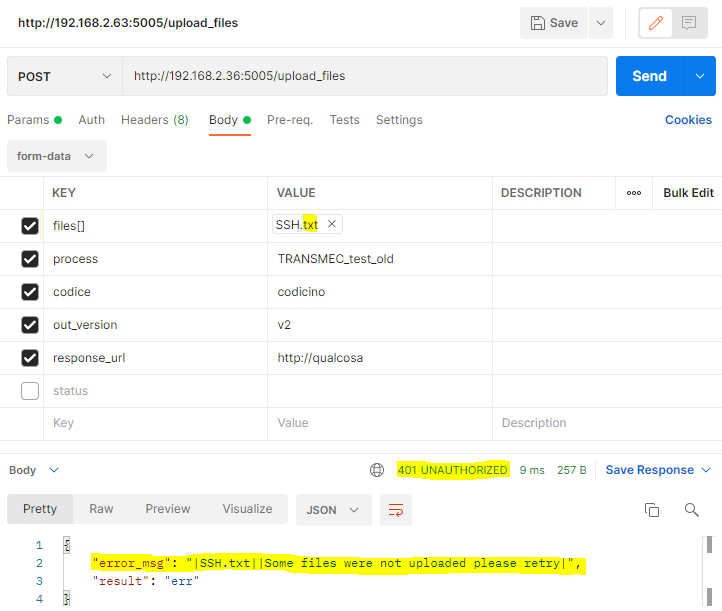

###### Caso 2

In questo caso manca un parametro nella chiamata.



###### Caso 3

In questo caso il file in allegato non ha l’estensione corretta.



## Codice Python – Process\_retriever

### Introduzione:

Il seguente servizio permette di controllare o verificare ogni minuto se ci sono dei file o lavori da elaborare, selezionare il preprocesso corrispondente e inviare il file importato alle api.

### Logica Process\_retriever.

Nel metodo run() ovvero nella fase di esecuzione del servizio è presente una query al database ‘import’ per verificare se ci sono dei nuovi import con stato “0”.

|  |  |  |
| --- | --- | --- |
| TABLE NAME | COLUMN\_NAME | Descrizione |
| Import | Id | Codice Identificativo import |
| Import | gr\_codice | Codice identificativo della sotto istanza (azienda del cliente) |
| Import | gr\_auth\_version | (input) versione dell’import:   * v1 * v2 * v3 |
| Import | gr\_response url | (input) Indirizzo dell’istanza a cui rispondere. |
| Import | nome\_file | (input) PATH/FILE |
| Import | status | Auto  0 = da elaborare (default, dopo l’import)  1 = in elaborazione  2 = elaborazione completata senza errori  3 = Error |

La lista dei processi attivi ha una capacità di massimo 4 processi, se il servizio rileva un nuovo file da elaborare e nella lista dei processi c’è almeno uno spazio il nuovo processo viene aggiunto nella lista dei processi attivi, viene lanciato in esecuzione **process\_launcher** e viene aggiornato lo stato dell’import da 0 a 1 ciò significa che è in fase di elaborazione.

#### Pre\_processor\_launcher():

In questa fase iniza l’elaborazione del file, gli vengono passati i dati dell’import e trova il preprocesso corrispondente a quello indicato, se non trova nessun preprocesso corrispondente viene aggiornato lo stato dell’import da 1 a 3 ciò significa che la fase di elaborazione non è andata a buon fine, in caso contrario viene selezionato il preprocesso corrispondente e invia il file alle api. Il preprocesso selezionato viene lanciato in esecuzione che trasforma il file importato in GR-friendly import file. Una volta terminato senza errori ritorna un risultato che contiene il file elaborato in output insieme alle credenziali del cliente per poter rispondere a greenrouter (il response\_url indica l’indirizzo dell’istanza a cui inviare l’output) che calcolerà poi la C02e e in fine lo stato dell’import s’aggiornerà da 1 a 2, in caso invece di errore durante l’elaborazione lo stato diventa 3.

#### Esempi:

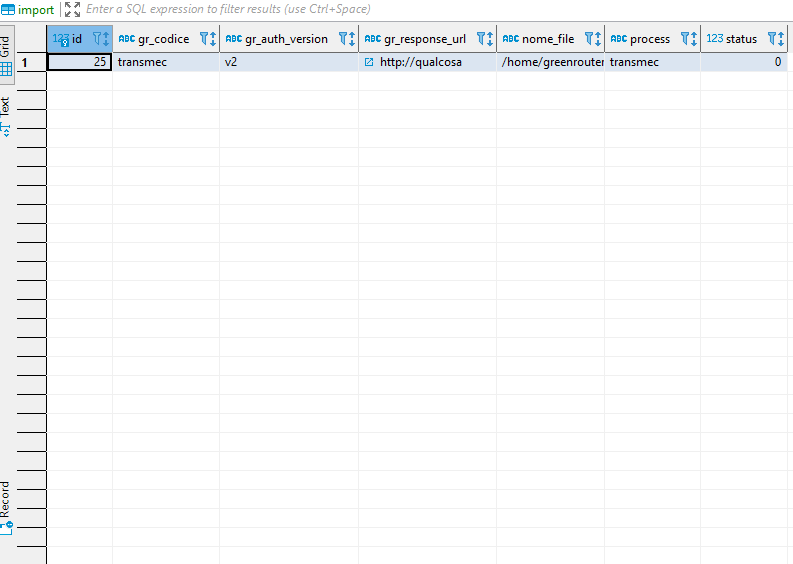
##### Non è stato rilevato nessun lavoro o file da elaborare.

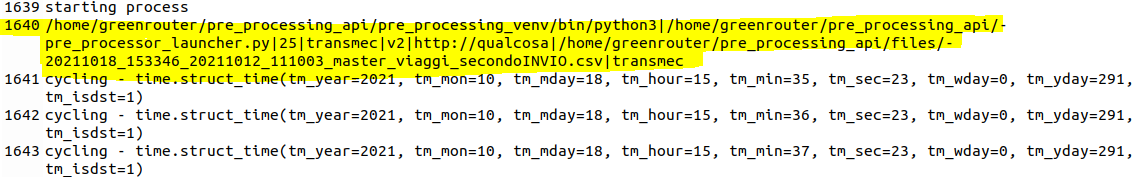
In questo caso abbiamo il servizio attivo ogni minuto controlla se ci sono dei nuovi file da elaborare ed aggiorna il suo log. In questo caso non c’è alcun file da elaborare.



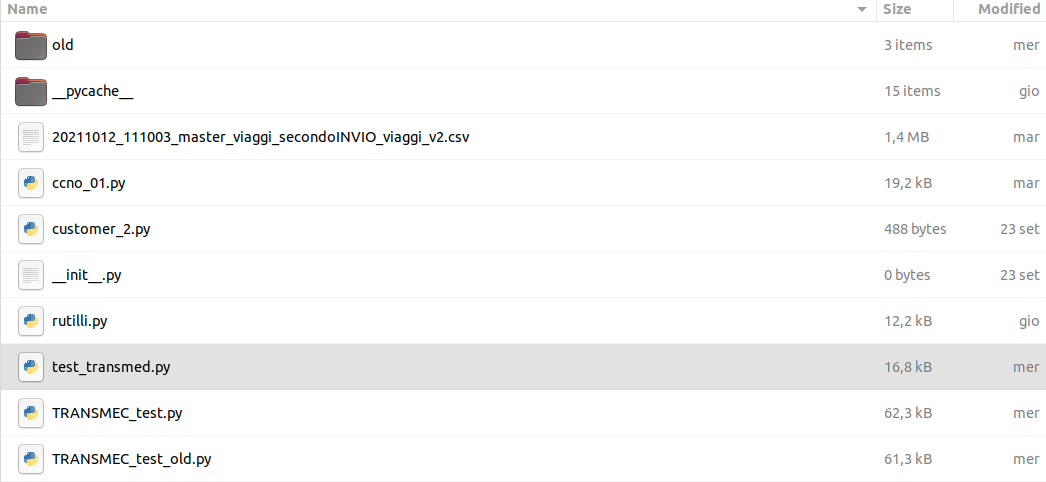
##### Nessun Pre-Processo trovato.

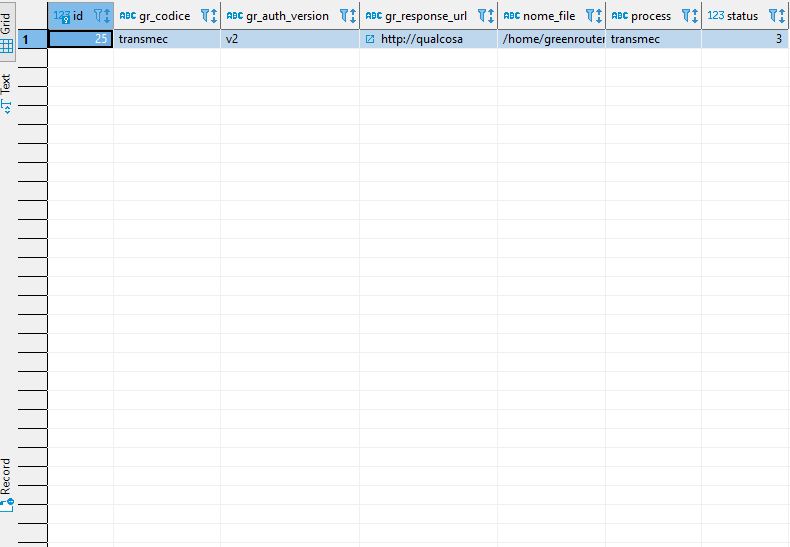
In questo caso è stato rilevato un nuovo lavoro o file da elaborare.



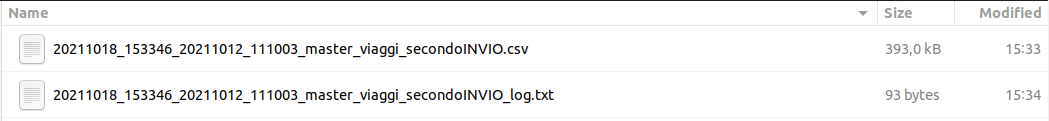


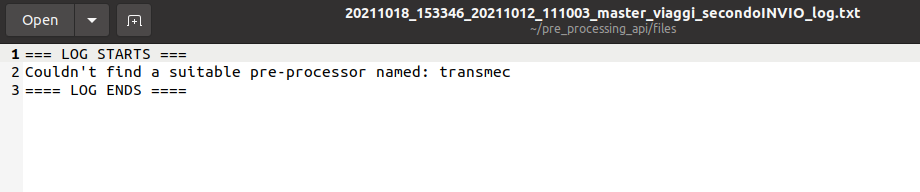
La fase successiva è lanciare in esecuzione il preprocesso indicato.  
Supponiamo che il processo indicato non esista.





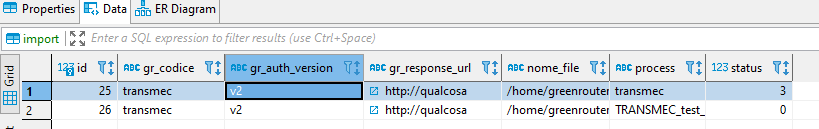
Lo stato si è aggiornato da 0 a 3 (0 🡪1🡪3), ciò significa che l’elaborazione non è andato a buon fine e possiamo vedere nel log che si trova nella stessa cartella nella quale è stato salvato o caricato il file.

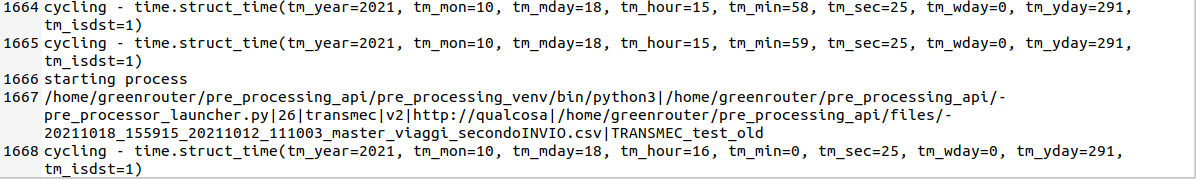




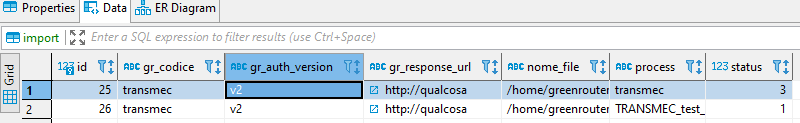
Vediamo che il pre-processo “transmec” non è stato trovato.

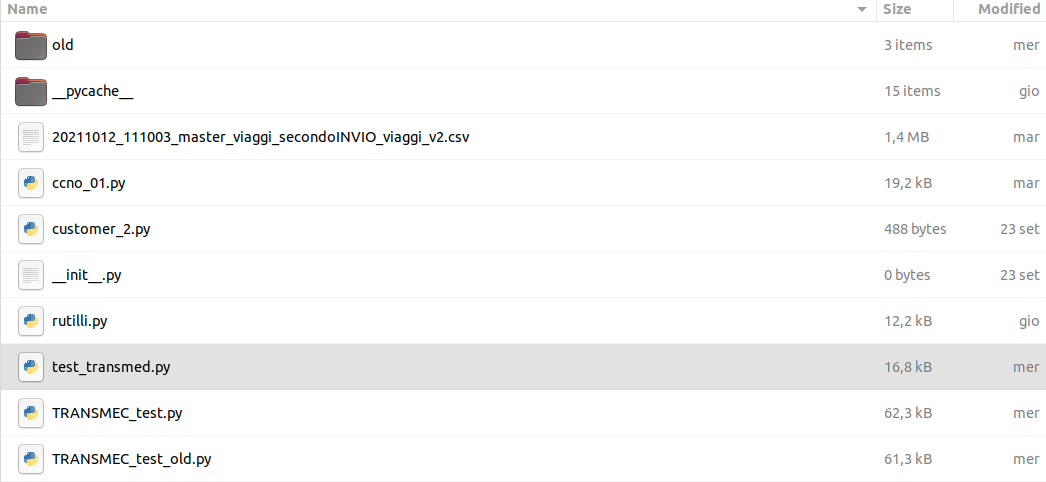
##### Pre-Processo trovato.



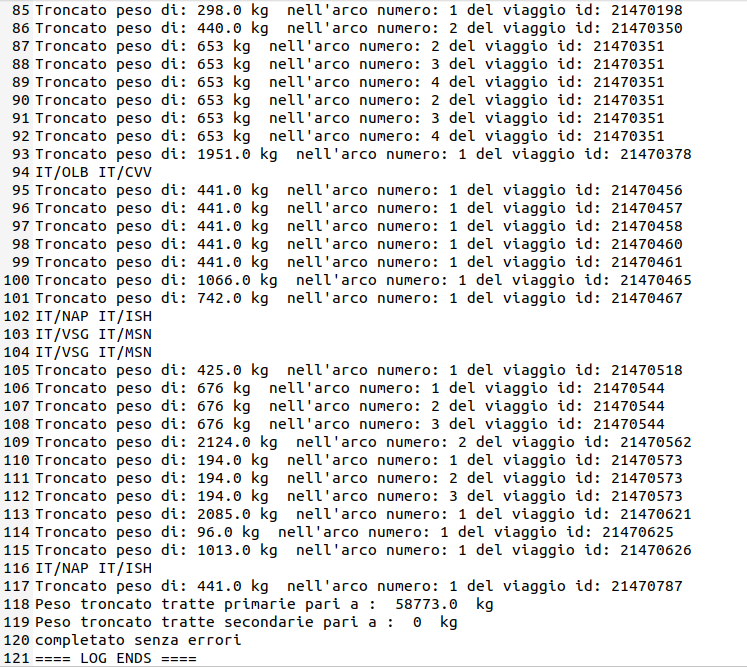
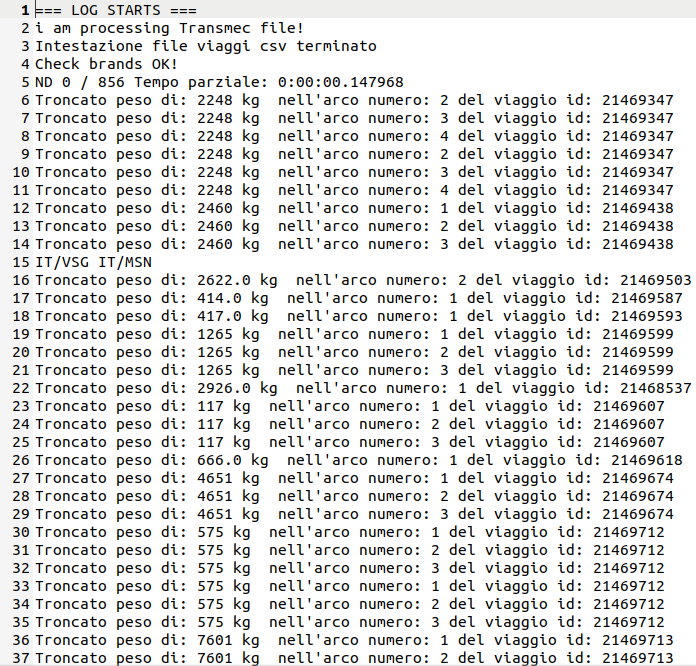


In questo caso è stato rilevato un nuovo lavoro o file da elaborare e il processo indicato esiste.



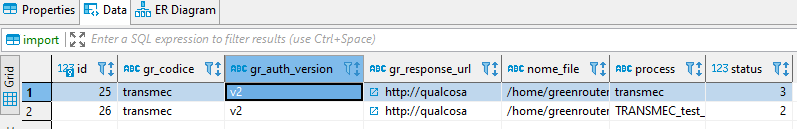


Il preprocesso corrispondente quindi viene lanciato in esecuzione e lo stato si aggiorna da 0 a 1 e   
possiamo vedere il log dell’elaborazione.

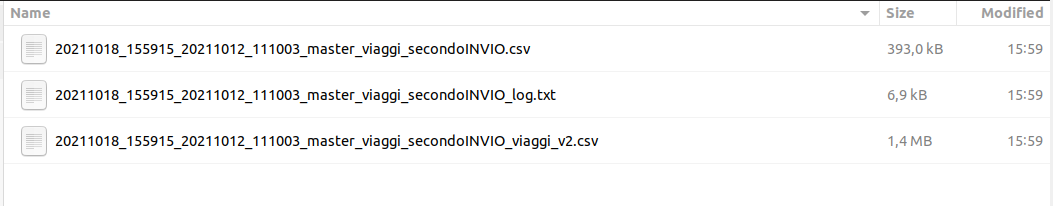


In questo caso il preprocesso ha funzionato perfettamente ed è stato completato senza errori.

Vediamo ora che lo stato è passato da 1 a 2.



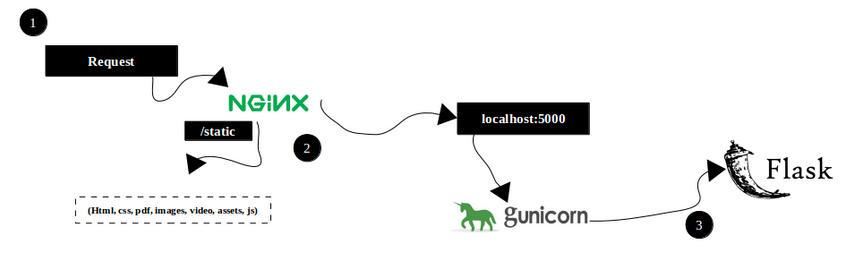
Oltre al log dell’elaborazione del file, possiamo trovare anche il file elaborato e generato dal preprocesso.



## Pre-Processing su Server Linux con NGINX Usando Gunicorn

### Introduzione:

* Nginx è un server (Web) http open source e un proxy inverso
* Gunicorn (Green Unicorn) è un server WSGI (Web Server Gateway Interface) Python che esegue il codice dell’applicazione web Python (Flask).



1. Le richieste sono gestite dal Web Server, in questo caso NGINX
2. Le richieste sono trasmesse poi a un server WSGI in questo caso Gunicorn .
3. Attraverso Gunicorn si può eseguire l’applicazione web, in questo caso Flask.

E’ stato utilizzato Gunicorn perché serve un protocollo di comunicazione tra il server web (Nginx) e un’applicazione web. Il server Web dovrebbe sapere come parlare con l’applicazione web e l’applicazione web dovrebbe sapere come rispondere al server web. Entrambi dovrebbero parlare usando “la stessa lingua” cioè “WSGI”.

### Configurazione:

#### Passaggio 1: Aggiornamenti

Prima di tutto sul server, sono stati installati tutti i pacchetti di cui abbiamo bisogno dai repository di Ubuntu:  
  
 $ sudo apt update

Sono state aggiornate le definizioni dei pacchetti e installati quelli che permetteranno di costruire l’ambiente Python, insieme ad alcuni altri pacchetti e strumenti di sviluppo:  
  
 $ sudo apt install python3-pip python3-dev build-essential libssl-dev libffi-dev python3- setuptools

#### Passaggio 2: Creazione di un ambiente virtuale Python

E’ stato configurato un ambiente virtuale per isolare le nostre applicazioni Flask dagli altri file python sul sistema.

Installazione del pacchetto “python3-venv”, che installerà il modulo “venv”:

$ sudo apt intall python3-venv

Abbiamo creato una directory principale per il nostro progetto Flask e poi ci siamo entrati.

$ mkdir ~/pre\_processing\_api

$ cd ~/pre\_processing\_api

Abbiamo Creato un ambiente virtuale per memorizzare i requisiti Python del progetto Flask:  
Installerà una copia locale di Python e pip in una directory chiamata pre\_processing\_venv all’interno della directory principale del progetto (pre\_processing\_api)

$ python3 -m venv pre\_processing\_venv

Abbiamo attivato poi l’ambiente virtuale.

$ source pre\_processing\_venv/bin/activate

#### Passaggio 3 – Configurazione dell’applicazioni Flask

Ora che siamo nel nostro ambiente virtuale, abbiamo installato Flask e Gunicorn e iniziato a proggettare le nostre applicazioni:

1. pre\_processing\_api\_v1
2. process\_retriever

Per prima cosa installiamo “wheel” con l’istanza locale di “pip” per garantire che i pacchetti vengano installati anche se mancano gli archives wheel.

$ pip install wheel

Successivamente abbiamo installato Flask, Gunicorn, flask-restful e altri moduli necessari per la creazione dell’applicazioni.

(pre\_processing\_venv) $ pip install gunicorn flask

Ora che abbiamo Flask disponibile abbiamo creato le due applicazioni.

(pre\_processing\_venv) $ nano ~/pre\_processing\_api/pre\_processing\_api\_v1.py

(pre\_processing\_venv) $ nano ~/pre\_processing\_api/process\_retriever.py

Il codice delle nostre applicazioni risiederà e questi file.

Per testare l’applicazione, è necessario consentire l’accesso alla porta “5000”:

$ sudo ufw allow 5000

Ora possiamo testare le app Flask:

(pre\_processing\_venv) $ python pre\_processing\_api\_v1.py

(pre\_processing\_venv) $ python process\_retriever.py

Vedremo un output come il seguente:

Output

\* Serving Flask app "pre\_processing\_api" (lazy loading)

\* Environment: production

WARNING: Do not use the development server in a production environment.

Use a production WSGI server instead.

\* Debug mode: off

\* Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)

Output

\* Serving Flask app "pre\_processing\_api" (lazy loading)

\* Environment: production

WARNING: Do not use the development server in a production environment.

Use a production WSGI server instead.

\* Debug mode: off

\* Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)

Possiamo ora provare a visitare l’indirizzo ip del nostro server seguito da “:5000” nel browser web, oppure provare le funzionalità delle nostre applicazioni per esempio le chiamate POST per “pre\_processing\_api\_v1” o vedere se “process\_retriever” gira senza errori.

**Creazione del punto di ingresso WSGI**

Abbiamo quindi, creato un file che servirà come punto di ingresso per la nostra applicazione “pre\_processing\_api\_v1”. Questo dirà al nostro server Gunicorn come interagire con l’applicazione.

Abbiamo chiamato il file “wsgi.py”:

(pre\_processing\_venv) $ nano ~/pre\_processing\_api/wsgi.py

In questo file, importiamo l’istanza Flask della nostra applicazione e poi la eseguiamo:

from pre\_processing\_api\_v1 import app

if \_\_name\_\_ == "\_\_main\_\_":

app.run()

Una volta finita salviamo e chiudiamo.

#### Passaggio 4 – Configurazione di Gunicorn

Prima di andare avanti, abbiamo verificato che Gunicorn possa servire corettamente l’applicazione.

E’ stato semplicemente passato il nome del punto di ingresso. Questo è costruito come il nome del modulo (meno il “.py“ interno), più il nome del chiamabile all’interno dell’applicazione. Nel nostro caso, questo è wsgi:app.

Abbiamo specificato anche l’interaccia e la porta a cui associare in modo che l’applicazione venga avviata su un’interfaccia disponibile pubblicamente:

(pre\_processing\_venv) $ cd ~/myproject

(pre\_processing\_venv) $ gunicorn –bind 0.0.0.0:5000 wsgi:app

Vedremo un output come il seguente:

Output

[2020-05-20 14:13:00 +0000] [46419] [INFO] Starting gunicorn 20.0.4

[2020-05-20 14:13:00 +0000] [46419] [INFO] Listening at: http://0.0.0.0:5000 (46419)

[2020-05-20 14:13:00 +0000] [46419] [INFO] Using worker: sync

[2020-05-20 14:13:00 +0000] [46421] [INFO] Booting worker with pid: 46421

Ora che abbiamo finito con l’ambiante virtuale, l’abbiamo disattivato:

(pre\_processing\_venv) $ deactivate

Tutti I comandi Python ora utilizzeranno l’ambiente Python del sistema.

Abbiamo quindi creato il file dell’unità di servizio systemd. La creazione di un file di unità systemd consentirà al sistema init di Ubuntu di avviare automaticamente Gunicorn e servire l’applicazione flask ogni volta che il server si avvia.

Creiamo quindi un file unitario che termina con .service all’interno del /etc/systemd/system directory per iniziare:

$ sudo nano /etc/system/system/pre\_processing\_api.service

All’interno inizieremo con la sezione [Unit], che viene utilizzata per specificare metadati e dipendenze. Abbiamo messo qui una descrizione del nostro servizio e comunichiamo al sistema init di avviarlo solo dopo che è stato raggiunto l’obbiettivo di rete:

[Unit]

Description=Gunicorn instance to serve pre\_processing\_api

After=network.target

Apriamo poi la sezione [Service]. Questo specificherà l’utente e il gruppo con cui vogliamo che il processo venga eseguito. Diamo al nostro account utente normale la proprietà del processo poiché possiede tutti i file pertinenti. Diamo anche la proprietà del gruppo al www-data group in modo che Nginx possa comunicare facilmente con i processi Gunicorn.

[Unit]

Description=Gunicorn instance to serve pre\_processing\_api

After=network.target

[Service]

User=greenrouter

Group=www-data

Quindi, abbiamo mappato la directory di lavoro e impostiamo il PATH variabile ambientale in modo che il sistema init sappia che gli eseguibili per il processo si trovano all’interno del nostro ambiente virtuale. Specifichiamo anche il comando per avviare il servizio. Questo comando farà quanto segue:

* Avvia 2 processi di lavoro
* Creato e assocciato a un file socket Unix, pre\_processing\_api.sock, all’interno della nostra directory di progetto. Abbiamo impostato un valore umask di 007 in modo che il file socket venga creato dando accesso al proprietario e al gruppo, limitando altri accessi.
* Specificato il nome del file del punto di ingresso WSGI, insieme al chiamabile Python all’interno di quel file (wsgi:app)
* Specificato l’ambiente o il path in cui salvare il log in caso di errori

Systemd richiede che diamo il percorso completo all’eseguibile Gunicorn, che è installato nel nostro ambiente virtuale.

[Unit]

Description=Gunicorn instance to serve pre\_processing\_api

After=network.target

[Service]

User=greenrouter

Group=www-data

WorkingDirectory=/home/greenrouter/pre\_processing\_api/

Environment="PATH=/home/greenrouter/pre\_processing\_api/pre\_processing\_venv/bin"

Infine, abbiamo aggiunto una sezione [Install]. Questo dirà a systemd a cosa collegare questo servizio se abilitiamo l’avvio all’avvio del server. Vogliamo che questo servizio venga avviato quando il normale sistema multiutente è attivo e funzionante:

[Unit]

Description=Gunicorn instance to serve pre\_processing\_api

After=network.target

[Service]

User=greenrouter

Group=www-data

WorkingDirectory=/home/greenrouter/pre\_processing\_api/

Environment="PATH=/home/greenrouter/pre\_processing\_api/pre\_processing\_venv/bin”

ExecStart=/home/greenrouter/pre\_processing\_api/pre\_processing\_venv/bin/gunicorn --workers 2 --bind unix:pre\_processing\_api.sock -m 007 wsgi:app --error-logfile /home/greenrouter/pre\_processing\_api/logExecStart=/home/greenrouter/pre\_processing\_api/pre\_processing\_venv/bin/gunicorn --workers 2 --bind unix:pre\_processing\_api.sock -m 007 wsgi:app --error-logfile /home/greenrouter/pre\_processing\_aps/gunicorn\_error.log --access-logfile /home/greenrouter/pre\_processing\_api/logs/gunicorn\_access.log --capture-output --log-level debug

[Install]

WantedBy=multi-user.target

Abbiamo poi configurato anche l’altro servizio per il process\_retriever (pre\_processing.service). “process\_retriever” diventerà un servizio a se stante su linux

[Unit]

Description= pre\_processing service

After=multi-user.target

[Service]

User=greenrouter

WorkingDirectory=/home/greenrouter/pre\_processing\_api/

Environment="PATH=/home/greenrouter/pre\_processing\_api/pre\_processing\_venv/bin"

ExecStart=/home/greenrouter/pre\_processing\_api/pre\_processing\_venv/bin/python3 /home/greenrouter/pre\_processing\_api/process\_retriever.py

[Install]

WantedBy=multi-user.target

Con ciò, i nostri file di servizio systemd è completo.

Ora possiamo avviare il servizio Gunicorn che abbiamo creato e abilitarlo in modo che inizi all’avvio:

$ sudo systemctl start pre\_processing\_api

$ sudo systemctl start pre\_processing

Controlliamo lo stato:

$ sudo systemctl status pre\_processing\_api

$ sudo systemctl status pre\_processing

Dovremo vedere un output come il seguente:

Output

● pre\_processing\_api.service - Gunicorn instance to serve pre\_processing\_api

Loaded: loaded (/etc/systemd/system/pre\_processing\_api.service; enabled; vendor preset: enabled)

Active: active (running) since Tue 2021-10-19 11:03:41 CEST; 22min ago

Main PID: 690 (gunicorn)

Tasks: 3 (limit: 9481)

Memory: 50.9M

CGroup: /system.slice/pre\_processing\_api.service

├─690 /home/greenrouter/pre\_processing\_api/pre\_processing\_venv/bin/python3 /home/greenrouter/pre\_processing\_api/pre\_processing\_venv/bin/gunicorn --workers 2 --bind unix:pre\_processing\_api.sock -m 007 wsgi:app --error-logfil>

├─825 /home/greenrouter/pre\_processing\_api/pre\_processing\_venv/bin/python3 /home/greenrouter/pre\_processing\_api/pre\_processing\_venv/bin/gunicorn --workers 2 --bind unix:pre\_processing\_api.sock -m 007 wsgi:app --error-logfil>

└─831 /home/greenrouter/pre\_processing\_api/pre\_processing\_venv/bin/python3 /home/greenrouter/pre\_processing\_api/pre\_processing\_venv/bin/gunicorn --workers 2 --bind unix:pre\_processing\_api.sock -m 007 wsgi:app --error-logfil

● pre\_processing.service - pre\_processing service

Loaded: loaded (/etc/systemd/system/pre\_processing.service; enabled; vendor preset: enabled)

Active: active (running) since Tue 2021-10-19 11:03:46 CEST; 23min ago

Main PID: 1376 (python3)

Tasks: 1 (limit: 9481)

Memory: 6.6M

CGroup: /system.slice/pre\_processing.service

└─1376 /home/greenrouter/pre\_processing\_api/pre\_processing\_venv/bin/python3 /home/greenrouter/pre\_processing\_api/process\_retriever.py

P.S. Se vediamo errore, dobbiamo assicurarci di risolverli prima di proseguire.

#### Passaggio 5 – Configurazione di Nginx per le richieste proxy

Il nostro server delle applicazioni Gunicorn dovrebbe ora essere attivo e funzionante, in attesa di richieste sul file socket nella directory del progetto. Configuriamo ora Nginx per passare le richieste web a quel socket apportando alcune piccole aggiunte al suo file di configurazione.

Abbiamo iniziato creando un nuovo file di configurazione del blocco del server in Nginx’s  
sites-available directory. Abbiamo chiamato questo pre\_processing\_api.

$ sudo nano /etc/nginx/sites-available/pre\_processing\_api

Abbiamo poi aperto un blocco server e comunicato a Nginx di ascoltare sulla porta predefinita 5005. E le diciamo anche di usare questo blocco per le richieste del nostro server:

Server {  
 listen 5005;  
 server\_name 172.21.10.20;  
}

Successivamente, abbiamo aggiunto un blocco di posizione che corrisponda a ogni richiesta. All’interno di questo blocco, abbiamo incluso il proxy\_params file che specifica alcuni parametri di proxy generali che devono essere impostati. Abbiamo quindi passato le richieste al socket che abbiamo definito usando il proxy\_pass direttiva:

Server {  
 listen 5005;  
 server\_name 172.21.10.20;  
 location / {  
 include proxy\_params;  
 proxy\_pass http://unix:/home/greenrouterpre\_processing\_api/pre\_processing\_api.sock;  
 }

}

Per abilitare la configurazione del blocco del server Nginx Che abbiamo creato, abbiamo collegato il file site-enabled:

$ sudo ln -s /etc/nginx/sites-available/pre\_processing\_api /etc/nginx/sites-enabled

Con il file in quella directory, abbiamo testato gli errori di sintassi:

$ sudo nginx -t

Se questo ritorna senza indicare alcun problema, riavviamo il processo Nginx per leggere la nuova configurazione:

$ sudo systemctl restart nginx

Infine, abbiamo regolato di nuovo il firewall. Non abbiamo più bisogno dell’accesso tramite porta 5000, quindi abbiamo rimosso quella regola. Abbiamo consentito l’accesso completo al server Nginx:

$ sudo ufw delete allow 5000

$ sudo ufw allow ‘Nginx Full’