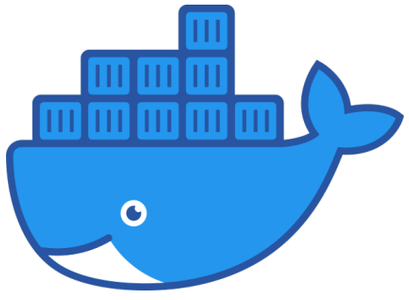
Carlo Lomello - 0124001651
Francesco Mabilia - 0124001910
Roberto Vecchio - 0124001871

CREAZIONE DI UN AMBIENTE DOCKER-COMPOSE

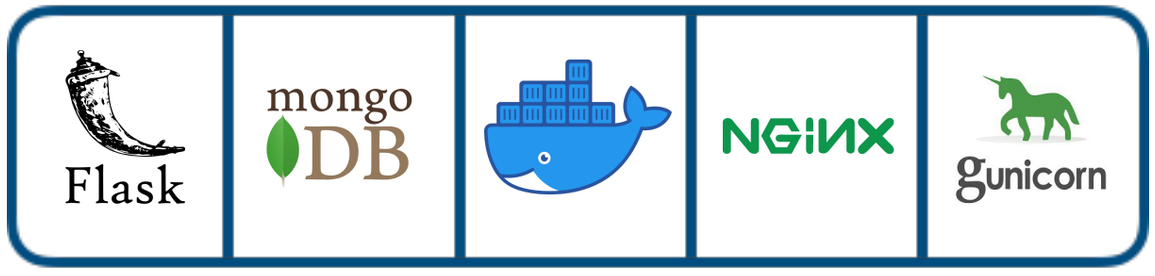


Lo sviluppo di applicazioni Web può diventare complesso e richiedere molto tempo durante la creazione e il mantenimento di diverse tecnologie.

Considerare delle opzioni più leggere, per ridurre la complessità e il tempo di produzione per l'applicazione, può risultare una soluzione più flessibile e scalabile.

Come micro framework web basato su Python, **Flask** offre agli sviluppatori un modo estensibile per far crescere le loro applicazioni attraverso estensioni che possono essere integrate nei progetti.

Stack delle tecnologie utilizzate



Carlo Lomello - 0124001651

Francesco Mabilia - 0124001910

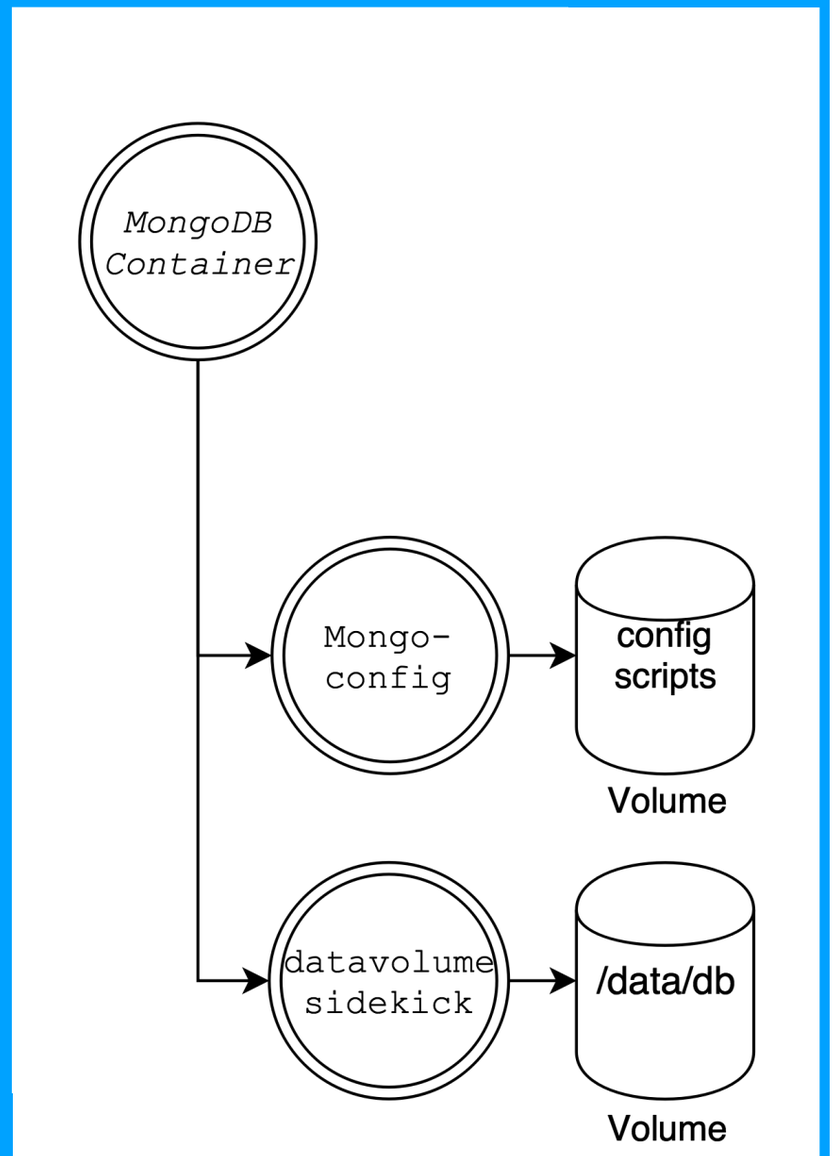
Roberto Vecchio - 0124001871

Mentre **MongoDB** è un database NoSQL progettato per adattarsi e funzionare con frequenti modifiche.

Inoltre gli sviluppatori possono utilizzare **Docker** per semplificare il processo di packaging e deploying delle loro applicazioni.

Docker Compose ha ulteriormente semplificato l'ambiente di sviluppo consentendo di definire la propria infrastruttura, inclusi i servizi delle applicazioni, i volumi di rete e i mount bind, in un unico file.

L'uso di Docker Compose fornisce facilità d'uso sull'esecuzione di più comandi di esecuzione del contenitore Docker.



Carlo Lomello - 0124001651

Francesco Mabilia - 0124001910

Roberto Vecchio - 0124001871

Andremo a creare, impacchettare ed eseguire l'applicazione Web con **Flask**, **Nginx** e **MongoDB** all'interno dei container **Docker.**

Definiamo l'intera configurazione dello stack in un file docker-compose.yml, insieme ai file di configurazione per i servizi connessi.

Flask richiede che un *web-server* soddisfi le richieste *HTTP*, quindi per utilizzare l'applicazione verrà utilizzato anche **Gunicorn**, che è un server ***HTTP WSGI Python***.

Nginx funge da *server proxy inverso* che inoltra le richieste a **Gunicorn** per l'elaborazione.

Carlo Lomello - 0124001651

Francesco Mabilia - 0124001910

Roberto Vecchio - 0124001871

Prerequisiti:

- Un utente non root con privilegi sudo configurati.
- Docker installato
- Un ide che permetti la modifica di testo (Visual Studio Code, PyCharm, etc…)

Carlo Lomello - 0124001651

Francesco Mabilia - 0124001910

Roberto Vecchio - 0124001871

Fase -1

L'infrastruttura può essere definita in un singolo file e creata con un singolo comando. In questo passaggio, imposteremo il file *docker-compose.yml* per eseguire l'applicazione **Flask**.

I servizi possono essere collegati tra loro e ognuno può avere un volume ad esso collegato per l'archiviazione persistente. I volumi sono memorizzati in una parte del *filesystem* host gestito da **Docker**.

I *volumi* sono il modo migliore per conservare i dati in **Docker**, poiché i dati nei volumi possono essere esportati o condivisi con altre applicazioni.

Per iniziare, crea una *directory* per l'applicazione nella home directory del tuo *pc/server*:

```
$ mkdir flaskapp
```

Passa alla directory appena creata:

```
$ cd flaskapp
```

Quindi, crea il file docker-compose.yml:

```
$ nano docker-compose.yml
```

Carlo Lomello - 0124001651

Francesco Mabilia - 0124001910

Roberto Vecchio - 0124001871

Carlo Lomello - 0124001651
Francesco Mabilia - 0124001910
Roberto Vecchio - 0124001871

1

```
version: ‘3.3’
services:

  flask:
    build:
      context: app
      dockerfile: Dockerfile
    container_name: flask
    image: digitalocean.com/flask-python:3.8
    restart: unless-stopped
    environment:
      APP_ENV: “development"
      APP_DEBUG: "True"
      APP_PORT: 5000
      MONGODB_DATABASE: flaskdb
      MONGODB_USERNAME: flaskuser
      MONGODB_PASSWORD: your_mongodb_password
      MONGODB_HOSTNAME: mongodb
    volumes:
      - ./app:/var/www
    depends_on:
      - mongodb
    networks:
      - frontend
      - backend
  mongodb:
    image: mongo:4.0.8
    container_name: mongodb
    restart: unless-stopped
    command: mongod --auth
    environment:
      MONGO_INITDB_ROOT_USERNAME: mongodbuser
      MONGO_INITDB_ROOT_PASSWORD: your_mongodb_root_password
      MONGO_INITDB_DATABASE: flaskdb
      MONGODB_DATA_DIR: /data/db
      MONDODB_LOG_DIR: /dev/null
    volumes:
      - mongodbdata:/data/db
    networks:
      - backend
```

2

```
  webserver:
    build:
      context: nginx
      dockerfile: Dockerfile
    image: digitalocean.com/webserver:latest
    container_name: webserver
    restart: unless-stopped
    environment:
      APP_ENV: "prod"
      APP_NAME: "webserver"
      APP_DEBUG: "true"
      SERVICE_NAME: "webserver"
    ports:
      - "80:80"
      - "443:443"
    volumes:
      - nginxdata:/var/log/nginx
    depends_on:
      - flask
    networks:
      - frontend

networks:
  frontend:
    driver: bridge
  backend:
    driver: bridge

volumes:
  mongodbdata:
    driver: local
  appdata:
    driver: local
  nginxdata:
    driver: local
```

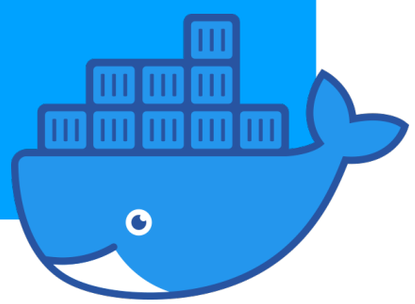
docker-compose.yml

Nel docker-compose.yml abbiamo i seguenti elementi:

- Un numero di versione che identifica la versione del file.
- La keyword *services:* viene utilizzata per elencare e definire tutti i servizi attivi nella nostra *web application.*

Definiamo ***flask*** come il primo servizio nel file docker-compose.yml:

- La proprietà ***build*** definisce il contesto della build. In questo caso, la cartella dell'app che conterrà il file Docker.
- Utilizzare la proprietà ***container_name*** per definire un nome per ciascun contenitore.
- La proprietà ***image*** specifica il nome dell'immagine e come verrà contrassegnata l'immagine Docker.
- La proprietà ***restart*** definisce il modo in cui il container deve essere riavviato.



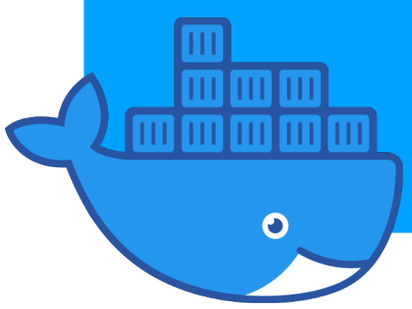
Carlo Lomello - 0124001651 **Francesco Mabilia - 0124001910** **Roberto Vecchio - 0124001871**

- La proprietà ***environment*** contiene le variabili di ambiente che vengono passate al contenitore.
  È necessario fornire una password sicura per la variabile di ambiente ***MONGODB_PASSWORD***.

- La proprietà ***volumes*** definisce i volumi che il servizio sta utilizzando. Nel nostro caso il volume ***appdata*** è montato all'interno del contenitore nella ***directory/var/www***.

- La proprietà ***depends_on*** definisce un servizio dal quale flask dipende per funzionare correttamente. In questo caso, il servizio ***flask*** dipenderà da *mongodb* poiché il servizio funge da **database** per l'applicazione.
  Inoltre garantisce che il servizio di ***flask*** sia in esecuzione solo se il servizio ***mongodb*** è in esecuzione.

- La proprietà ***networks*** specifica *frontend* e *backend* come reti a cui avrà accesso flask.



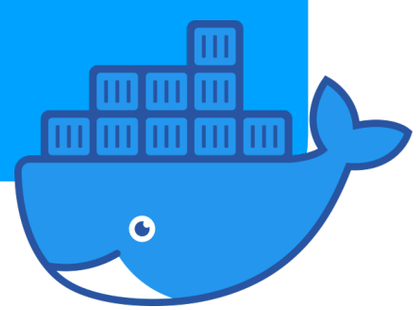
Successivamente, andremo a definire il ***MongoDB*** per la tua applicazione:

- ***Container_name***: per questo servizio è *mongodb* con una politica di riavvio a meno che non sia stata arrestata.

- Utilizzare la proprietà ***command*** per definire il comando che verrà eseguito all'avvio del contenitore. Il comando *mongod --auth* disabiliterà l'accesso alla shell MongoDB senza credenziali, il che proteggerà il database richiedendo l'autenticazione.

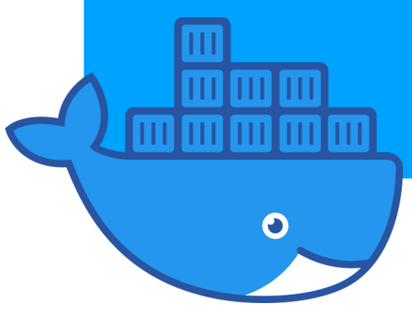
- Le variabili di ambiente ***MONGO_INITDB_ROOT_USERNAME*** e ***MONGO_INITDB_ROOT_PASSWORD*** creano un utente root con le credenziali fornite, quindi assicurati di sostituire il placeholder con una password complessa.

- MongoDB memorizza i suoi dati in */data/db* per impostazione predefinita, quindi i dati nella cartella */data/db* verranno scritti nel volume denominato *mongodbdata* per la persistenza. Di conseguenza, non perderai i tuoi database in caso di riavvio. Il servizio ***mongoDB*** non espone alcuna porta, quindi esso sarà accessibile solo attraverso la rete di *backend*.



Ora andiamo a definire il ***web server*** per la tua applicazione:

- Qui hai definito il contesto della build, che è la cartella ***nginx*** contenente il ***Dockerfile***.
- Con la proprietà ***image***, si specifica l'immagine utilizzata per contrassegnare ed eseguire il contenitore.
- La proprietà ***ports*** configurerà il servizio *Nginx* in modo che sia accessibile pubblicamente tramite le porte *: 80* e *: 443*
- ***volumes*** monta il volume *nginxdata* all'interno del contenitore nella directory */var/log/nginx*.
- Con ***depends_on*** definiamo la dipendenza del *web server* da *flask*
- Infine, la proprietà ***networks*** definisce il network di appartenenza



Successivamente, creeremo dei ***networks-bridge*** per consentire ai container di comunicare tra loro.

Sono state definite due reti, ***frontend*** e ***backend***, per i servizi a cui connettersi. I servizi *frontend*, come **Nginx**, si collegheranno alla rete *frontend* poiché deve essere accessibile pubblicamente. I servizi di *backend*, come **MongoDB**, si collegheranno alla rete di *backend* per impedire l'accesso non autorizzato al servizio.

Successivamente, utilizzeremo i *volumi* per rendere persistenti i *file* e le *modifiche* apportate di *configurazione*, al *database* e all'*applicazione.* Questi ultimi, sono gestiti da ***Docker*** e archiviati nel *filesystem*.

Nella sezione ***volumes*** andremo a dichiarare i volumi che l'applicazione utilizzerà per conservare i dati. Qui sono stati definiti i volumi *mongodbdata*, *appdata* e *nginxdata* per il persistere dei database ***MongoDB***, dei dati dell'applicazione *Flask* e dei *log* del *web server* ***Nginx***. Tutti questi volumi utilizzano un driver locale per archiviare i dati localmente. I volumi vengono utilizzati per conservare informazioni in modo tale che i *dati* dei *database* e dei *log* del *web server,* possano evitare di essere persi dopo il riavvio dei contenitori.

Carlo Lomello - 0124001651

Francesco Mabilia - 0124001910

Roberto Vecchio - 0124001871

Fase - 2 scrivere i file Docker di flask e del server Web

Con Docker, è possibile creare contenitori per eseguire le applicazioni da un file chiamato Dockerfile.

Dockerfile è uno strumento che consente di creare immagini personalizzate, il quale è possibile utilizzare per installare il software richiesto dall'applicazione e quindi configurare i contenitori in base alle proprie esigenze.

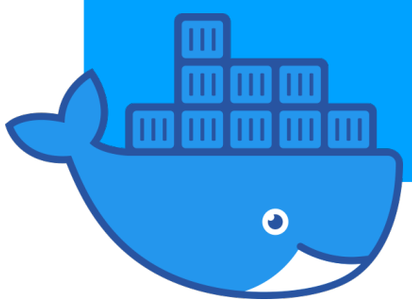
In questo passaggio, scriveremo il Dockerfile per il servizi Flask.

Per iniziare, creiamo la directory dell'app:

```
$ mkdir app
```

Quindi, crea il Dockerfile nella directory:

```
$ nano app/Dockerfile
```

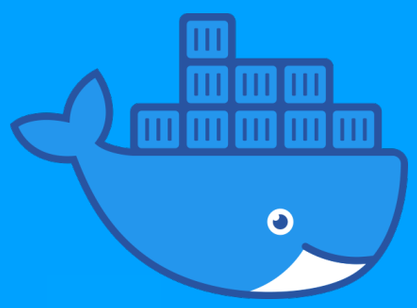


Carlo Lomello - 0124001651 **Francesco Mabilia - 0124001910** **Roberto Vecchio - 0124001871**

Carlo Lomello - 0124001651
Francesco Mabilia - 0124001910
Roberto Vecchio - 0124001871

```
FROM python:3.7.7-alpine3.10

LABEL MAINTAINER="FirstName LastName”
ENV GROUP_ID=1000 \
    USER_ID=1000

WORKDIR /var/www/

ADD . /var/www/
RUN pip install -r requirements.txt
RUN pip install gunicorn

RUN addgroup -g $GROUP_ID www
RUN adduser -D -u $USER_ID -G www www -s /bin/sh

USER www

EXPOSE 5000

CMD [ "gunicorn", "-w", "4", "--bind", "0.0.0.0:5000", "wsgi"]
```

app/Dockerfile

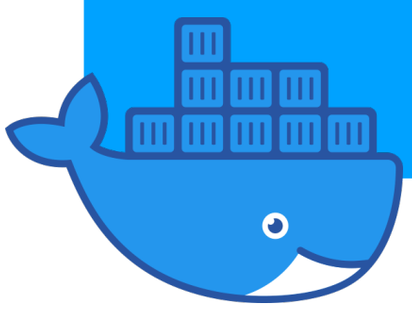


Attraverso questo ***Dockerfile***, verrà scaricata un'immagine *3.6.8-alpine3.9* con *Python 3.6.8* preinstallato.

La direttiva *ENV* viene utilizzata per definire le variabili di ambiente per il nostro ***GROUP_ID*** e ***USER_ID***

Linux Standard Base (LSB) specifica che *UID* e *GID* 0-99 sono allocati staticamente dal sistema. Gli *UID* 100-999 dovrebbero essere allocati dinamicamente per utenti e gruppi di sistema. Gli *UID* 1000-59999 dovrebbero essere allocati dinamicamente per gli account utente. Tenendo presente questo, è possibile assegnare in modo sicuro un *UID* e un *GID* di 1000.

La direttiva ***WORKDIR*** definisce la *directory* di lavoro per il contenitore. Assicuriamoci di sostituire il campo ***LABEL MAINTAINER*** con il tuo nome e indirizzo email.



Carlo Lomello - 0124001651

Francesco Mabilia - 0124001910

Roberto Vecchio - 0124001871

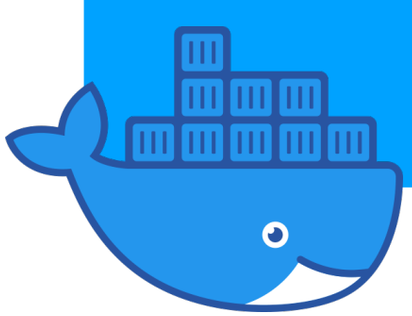
La direttiva ***ADD*** viene utilizzata per copiare i file dalla *directory* dell'app locale in quella /var/www sul contenitore. Per impostazione predefinita, i contenitori Docker vengono eseguiti come utente root.

L'utente root ha accesso a tutto nel sistema, quindi le implicazioni di una violazione della sicurezza possono essere disastrose. Per mitigare questo rischio per la sicurezza, ciò creerà un nuovo utente e gruppo che avrà accesso solo alla directory */var/www*.

Successivamente, ***Dockerfile*** utilizzerà la direttiva ***RUN*** per installare *Gunicorn* e i pacchetti specificati nel file requirements.txt, che verrà creato più avanti.

Il comando ***USER*** definisce che i programmi eseguiti nel contenitore useranno l'utente *www*.

Gunicorn si metterà in ascolto sulla porta: 5000, quindi aprirà questa porta con il comando ***EXPOSE***.



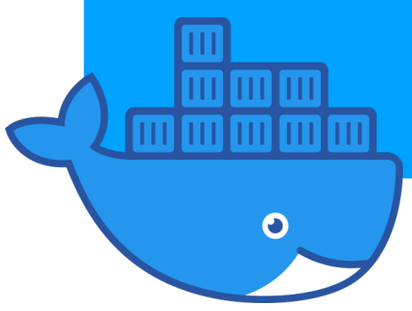
Carlo Lomello - 0124001651

Francesco Mabilia - 0124001910

Roberto Vecchio - 0124001871

La riga CMD ["gunicorn", "-w", "4", "--bind", "0.0.0.0:5000", "wsgi"] esegue il comando per avviare il server Gunicorn con quattro operatori in ascolto su porta 5000. Il numero dovrebbe generalmente essere compreso tra 2 e 4 lavoratori per core nel server, la documentazione di Gunicorn consiglia (2 x $ num_cores) + 1 come numero di worker con cui iniziare.

Infine, viene quindi aggiunto un nuovo utente e gruppo ed inizializzata l’applicazione.



Carlo Lomello - 0124001651

Francesco Mabilia - 0124001910

Roberto Vecchio - 0124001871

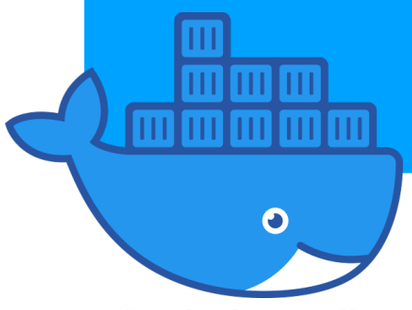
Quindi, crea una nuova directory per contenere la tua configurazione Nginx:

```
$ mkdir nginx
```

Quindi creare il Dockerfile per il server Web Nginx nella directory nginx:

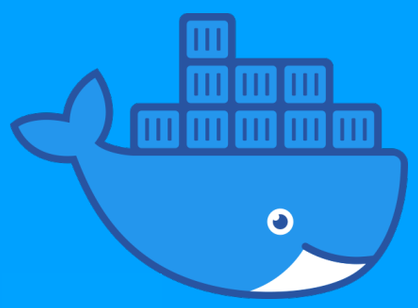
```
$ nano nginx/Dockerfile
```

Aggiungi il seguente codice al file per creare il Dockerfile che costruirà l'immagine per il tuo contenitore Nginx:



Carlo Lomello - 0124001651
Francesco Mabilia - 0124001910
Roberto Vecchio - 0124001871

```
FROM digitalocean.com/alpine:latest

LABEL MAINTAINER="FirstName LastName”
RUN apk --update add nginx && \
    ln -sf /dev/stdout /var/log/nginx/access.log && \
    ln -sf /dev/stderr /var/log/nginx/error.log && \
    mkdir /etc/nginx/sites-enabled/ && \
    mkdir -p /run/nginx && \
    rm -rf /etc/nginx/conf.d/default.conf && \
    rm -rf /var/cache/apk/*

COPY conf.d/app.conf /etc/nginx/conf.d/app.conf

EXPOSE 80 443
CMD ["nginx", "-g", "daemon off;"]
```



nginx/Dockerfile

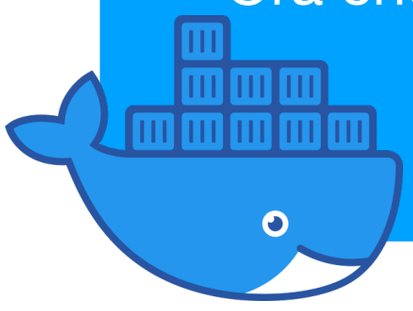
Questo file Docker di Nginx utilizza un'immagine di base alpine,
che è una piccola distribuzione Linux.
Nella direttiva RUN si installa nginx e si creano collegamenti simbolici per pubblicare l'errore e accedere ai registri sullo standard error (/dev/stderr) e sull'output (/dev/stdout).

Al termine, vengono eseguiti i comandi per rimuovere *default.conf* e */var/cache/apk/** per ridurre le dimensioni dell'immagine risultante. L'esecuzione di tutti questi comandi in un singolo ***RUN*** riduce il numero di livelli nell'immagine, riducendo anche le dimensioni dell'immagine risultante.

La direttiva ***COPY***, copia la configurazione del *web server app.conf* all'interno del contenitore.

La direttiva ***EXPOSE*** garantisce che i *container* siano in ascolto sulle porte: *80 e: 443*, poiché l'applicazione verrà eseguita su: *80* con: *443* come porta protetta.

Infine, la direttiva ***CMD*** definisce il comando per avviare il *server Nginx*.
Ora che ***Dockerfile*** è pronto, è possibile configurare il *proxy inverso Nginx* per instradare il traffico all'applicazione *Flask*.



Carlo Lomello - 0124001651

Francesco Mabilia - 0124001910

Roberto Vecchio - 0124001871

Fase - 3 configurazione del proxy inverso Nginx

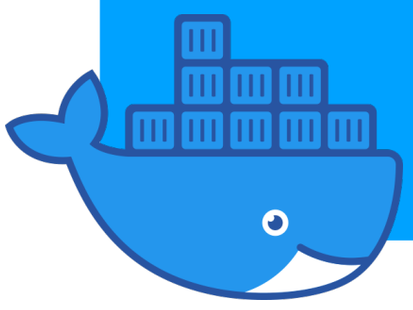
In questo passaggio, configureremo **Nginx** come *proxy inverso* per inoltrare richieste a **Gunicorn** su: 5000. Un *server proxy inverso* viene utilizzato per indirizzare le richieste *client* al *server back-end* appropriato. Questo fornisce un ulteriore livello di astrazione e controllo per garantire il flusso regolare del traffico di rete tra *client* e *server*.

Inizia creando la directory *nginx/conf.d*:

```
$ mkdir nginx/conf.d
```

Per configurare **Nginx**, è necessario creare un file *app.conf* nella cartella *nginx/conf.d/*.
Il file *app.conf* contiene la configurazione necessaria al proxy inverso per inoltrare le richieste a ***Gunicorn***.

```
$ nano nginx/conf.d/app.conf
```



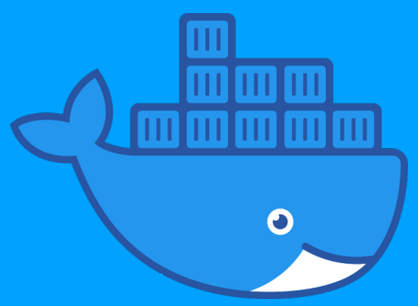
Carlo Lomello - 0124001651

Francesco Mabilia - 0124001910

Roberto Vecchio - 0124001871

Carlo Lomello - 0124001651
Francesco Mabilia - 0124001910
Roberto Vecchio - 0124001871

```
upstream app_server {
    server flask:5000;
}

server {
    listen 80;
    server_name _;
    error_log  /var/log/nginx/error.log;
    access_log /var/log/nginx/access.log;
    client_max_body_size 64M;

    location / {
        try_files $uri @proxy_to_app;
    }

    location @proxy_to_app {
        gzip_static on;

        proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
        proxy_set_header X-Forwarded-Proto $scheme;
        proxy_set_header Host $http_host;
        proxy_buffering off;
        proxy_redirect off;
        proxy_pass http://app_server;
    }
}
```



nginx/conf.d/app.conf

Ciò definirà innanzitutto il *server upstream*, che viene comunemente utilizzato per specificare un *web server* o *app* per il *routing*.

Il server *upstream app_server*, definisce l'indirizzo del *server* con la direttiva *server*, identificata dal nome del contenitore *flask:5000*.

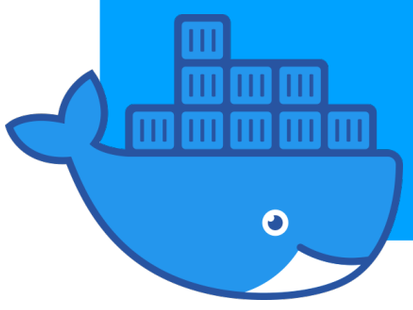
La configurazione per il *web server Nginx* è definita nel *blocco server*.

La direttiva ***listen*** definisce il numero di porta su cui il server ascolterà le richieste in arrivo.

Le direttive ***error_log*** e ***access_log*** definiscono i file per la scrittura dei registri.

La direttiva ***proxy_pass*** viene utilizzata per impostare il *server upstream* per l'inoltro delle richieste a *http://app_server*.

Con il *web server Nginx* configurato, è possibile passare alla creazione dell'***API Flask***.



Carlo Lomello - 0124001651

Francesco Mabilia - 0124001910

Roberto Vecchio - 0124001871

Fase - 4 creazione dell'API Flask

Ora che abbiamo creato il nostro ambiente, siamo pronti per la creazione dell’applicazione vera e propria!

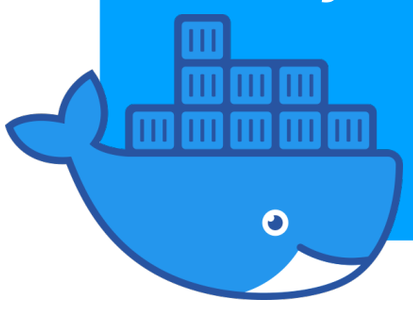
In questo passaggio scriveremo un'applicazione che ci consentirà di effettuare una registrazione utente inserendo dei dati ed un’authentication.

Iniziamo creando il file requirements.txt nella directory dell'app:

```
$ nano app/requirements.txt
```

Questo file viene utilizzato per installare le dipendenze per l'applicazione. L'implementazione di questo tutorial utilizzerà:

- ***Flask***
- ***Flask-PyMongo***
- ***request***
- ***Jinja2***



Carlo Lomello - 0124001651

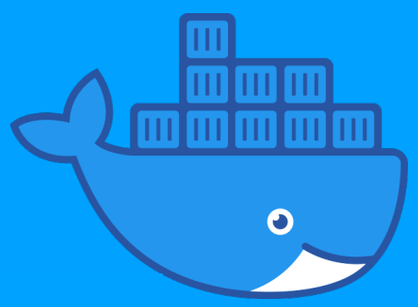
Francesco Mabilia - 0124001910

Roberto Vecchio - 0124001871

Carlo Lomello - 0124001651
Francesco Mabilia - 0124001910
Roberto Vecchio - 0124001871

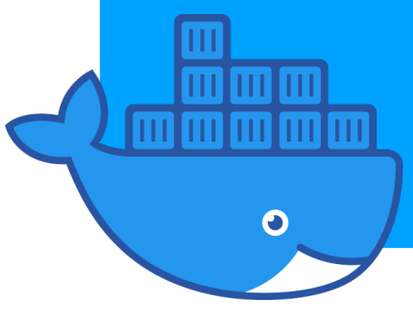
```
Flask==1.1.1
Flask-PyMongo==2.3.0
requests==2.20.1
Jinja2==2.11.1
```

app/requirements.txt



Quindi, creiamo il file *app.py* per contenere il codice dell'applicazione Flask nella directory dell'app:

```
$ nano app/app.py
```



Carlo Lomello - 0124001651

Francesco Mabilia - 0124001910

Roberto Vecchio - 0124001871

Carlo Lomello - 0124001651 **Francesco Mabilia - 0124001910** **Roberto Vecchio - 0124001871**

1

```
from flask import Flask, request, render_template
from flask_pymongo import PyMongo
import os

application = Flask(__name__, static_url_path='', static_folder='static')

mongo_conn_uri = 'mongodb://' +\
    os.environ['MONGODB_USERNAME'] +\
    ':' + os.environ['MONGODB_PASSWORD'] +\
    '@' + os.environ['MONGODB_HOSTNAME'] +\
    ':27017/' + os.environ['MONGODB_DATABASE']

application.config["MONGO_URI"] = mongo_conn_uri
mongo = PyMongo(application)
db = mongo.db

@application.route("/")
def index():
    return render_template("WebContent/homepage.html")

@application.route("/register")
def register():
    return render_template("WebContent/register_page.html")

@application.route('/register_conf')
def register_conf():
    username = request.args.get("username")
    password = request.args.get("password")
    nome = request.args.get("nome")
    cognome = request.args.get("cognome")
    telefono = request.args.get("telefono")

    item = {
        'username': username,
        'password': password,
        'nome': nome,
        'cognome': cognome,
        'telefono': telefono
    }
    db.persona.insert_one(item)

    return render_template("WebContent/homepage.html")
```

2

```
@application.route("/login")
def login():
    return render_template("WebContent/login_page.html")

@application.route('/logged_in')
def logged_in():
    username = request.args.get("username")
    password = request.args.get("password")

    _persona = db.persona.find({
        'username': username,
        'password': password
        })

    item = {}
    data = []

    for pers_temp in _persona:
        item = {
            'nome': pers_temp['nome'],
            'cognome': pers_temp['cognome'],
            'telefono': pers_temp['telefono']
        }
        data.append(item)

    for pers_temp in data:
        nome = pers_temp['nome']
        cognome = pers_temp['cognome']
        telefono = pers_temp['telefono']

    return render_template('WebContent/welcome.html', nome = nome,
                           cognome = cognome, telefono = telefono)

if __name__ == "__main__":
    ENVIRONMENT_DEBUG = os.environ.get("APP_DEBUG", True)
    ENVIRONMENT_PORT = os.environ.get("APP_PORT", 5000)
    application.run(host='0.0.0.0', port=ENVIRONMENT_PORT,  debug=ENVIRONMENT_DEBUG)
```

app/app.py

- Prima di tutto importiamo Flask, request e render_template dal framework flask e successivamente importiamo PyMongo da flask_pymongo, il tutto per poter utilizzare i metodi necessari al corretto funzionamento della nostra applicazione, quindi per poter interagire con le pagine html, configurare la connessione al nostro database MongoDB e per poter scrivere e leggere informazioni su di esso. Infine viene importato os per interagire e ottenere informazioni sul sistema operativo.

- L’istanza application è un oggetto della classe Flask. L’unico argomento richiesto dal costruttore della classe Flask è il nome del principale modulo o package dell’applicazione. Quindi Flask usa il suo argomento per determinare il root path dell’applicazione così potrà in un secondo momento utilizzarlo per trovare file di risorse relativi alla location dell’applicazione.

Carlo Lomello - 0124001651 **Francesco Mabilia - 0124001910** **Roberto Vecchio - 0124001871**

La variabile mongo_conn_uri è una stringa che concatena ad essa una serie di informazioni volte ad identificare in univocamente l'indirizzo del nostro database. Composto quindi da:

- Protocollo
- Username Database
- Password database
- Nome Host
- Porta
- Nome del database

Successivamente viene associato all'attributo MONGO_URI la nostra variabile mongo_conn_uri.

Di seguito viene istanziato un oggetto della classe PyMongo e viene passata come argomento l'istanza della classe Flask (application).

Infine con la variabile db viene associata la connessione al nostro servizio database.

Carlo Lomello - 0124001651 **Francesco Mabilia - 0124001910** **Roberto Vecchio - 0124001871**

- Con la view function ***index( )*** viene renderizzata una view che corrisponde al codice html scritto nel file homepage.html locato nella folder *templates/src*.

- Con la view function ***register( )*** viene renderizzata una view che corrisponde al codice html scritto nel file register_page.html locato nella folder *templates/src*.

- Con la view function ***login( )*** viene renderizzata una view che corrisponde al codice html scritto nel file login_page.html locato nella folder *templates/src*.

Carlo Lomello - 0124001651

Francesco Mabilia - 0124001910

Roberto Vecchio - 0124001871

- La view function ***register_conf( )*** preleva le informazioni mediante una **form** e le inseriamo nel nostro database sotto la collection ***persona.***
  Dopodiché la pagina verrà reindirizzata alla pagina homepage.html

- La view function ***logged_in( )*** preleva le informazioni di accesso mediante una **form**, preleva i dati relativi all'utente dal nostro database ed infine la pagina verrà reindirizzata alla *welcome.html* dove verranno visualizzate le informazioni relative alla *persona*.

- L'istanza *application* ha bisogno di eseguire un metodo che lancia il web server Flask. Una volta che il server è startato, entra in un loop che attende per richieste e servizi e continua fino a che l'applicazione non viene stoppata, come ad esempio premendo ctrl+c da terminale.

Carlo Lomello - 0124001651 **Francesco Mabilia - 0124001910** **Roberto Vecchio - 0124001871**

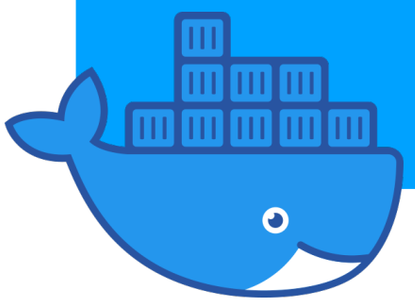
Andiamo ora a creare il file ***wsgi.py*** nella directory dell'app.

```
$ nano app/wsgi.py
```

Il file ***wsgi.py*** crea un oggetto application in modo che il server possa utilizzarlo. Ogni volta che arriva una richiesta, il server utilizza questo oggetto application per eseguire i gestori di richieste dopo aver analizzato l'URL.

```
from app import application

if __name__ == "__main__":
  application.run()
```

app/wsgi.py



Carlo Lomello - 0124001651

Francesco Mabilia - 0124001910

Roberto Vecchio - 0124001871

Ora possiamo creare la folder ***templates*** che andrà ad ospitare i nostri *contenuti web*.

```
$ mkdir app/templates
```

All'interno di questa cartella, *Flask* utilizzerà i template di *Jinja2* (un motore che permette l'utilizzo del linguaggio *Python* all'interno di un file *html*).

In templates, creeremo a nostra volta altre due cartelle per organizzare il contenuto. Quindi andiamo a creare la folder ***templatesFolder*** che conterrà i *template* delle nostre pagine web e poi la folder ***WebContent*** che conterrà le nostre *pagine html*

```
$ mkdir app/templates/templatesFolder
```

```
$ mkdir app/templates/WebContent
```

Carlo Lomello - 0124001651 **Francesco Mabilia - 0124001910** **Roberto Vecchio - 0124001871**

Il nostro progetto avrà un ***template*** di default che andremo ad utilizzare per tutte le nostre *pagine html*

```
$ nano app/templates/templatesFolder/project_template.html
```

Carlo Lomello - 0124001651

Francesco Mabilia - 0124001910

Roberto Vecchio - 0124001871

Carlo Lomello - 0124001651 Francesco Mabilia - 0124001910 Roberto Vecchio - 0124001871

1

```
<!DOCTYPE html>
<html>
   <head>
      <meta charset="UTF-8">
      {% block meta %}
      {% endblock %}
      <script src="https://code.jquery.com/jquery-3.4.1.slim.min.js"
            integrity="sha384-
J6qa4849blE2+poT4WnyKhv5vZF5SrPo0iEjwBvKU7imGFAV0wwj1yYfoRSJoZ+n"
            crossorigin="anonymous"></script>
      <script src="https://cdn.jsdelivr.net/npm/popper.js@1.16.0/dist/umd/popper.min.js"
            integrity="sha384-
Q6E9RHvbIyZFJoft+2mJbHaEWldlvI9IOYy5n3zV9zzTtmI3UksdQRVvoxMfooAo"
            crossorigin="anonymous"></script>
      <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/
css/bootstrap.min.css"
            integrity="sha384-Vkoo8x4CGsO3+Hhxv8T/
Q5PaXtkKtu6ug5TOeNV6gBiFeWPGFN9MuhOf23Q9Ifjh"
            crossorigin="anonymous">
      <script src="https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/js/
bootstrap.min.js"
            integrity="sha384-
wfSDF2E50Y2D1uUdj0O3uMBJnjuUD4Ih7YwaYd1iqfktj0Uod8GCExl3Og8ifwB6"
            crossorigin="anonymous"></script>
      <link rel="stylesheet" type="text/css" href="{{ url_for('static', filename='css/
project_style.css') }}">

      {% block head %}
      {% endblock %}
   </head>
   <body class="d-flex flex-column”>
      <header>
         <div class="container" >
            <div class="row">
```

2

```
               <div class="col-12">
                  <div class="text-center">
                     <a href="{{url_for('index')}}">
                        <img src="{{ url_for('static', filename='assets/docker-brands.svg') }}”
class="mt-5">
                     </a>
                  </div>
               </div>
            </div>
            {% block header %}
            {% endblock %}
         </header>
         <div id="page-content">
            <div class="container text-center">
               <div class="row justify-content-center">
                  <div class="col-md-7">
                     {% block body %}
                     {% endblock %}
                  </div>
               </div>
            </div>
         </div>
         <footer id="sticky-footer" class="py-4 bg-dark text-white-50">
          <div class="container text-center d-flex justify-content-around">
            <small>Carlo Lomello - 0124001651</small>
            <small>Francesco Mabilia - 0124001910</small>
            <small>Roberto Vecchio - 0124001871</small>
         </div>
            {% block footer %}
            {% endblock %}
         </footer>
      </body>
      {% block other %}
      {% endblock %}
</html>
```

Jinja

templates/templatesFolder/project_template.html

HTML

La pagina ***project_template.html*** funge da base per lo sviluppo della struttura e dello stile / interazione di altre pagine e presenta il boilerplate html, ovvero:

- head - dove abbiamo incluso la libreria bootstrap ed il nostro foglio di stile generico. Inoltre, *jinja2* attraverso la sintassi {% block nome_block %}…{% endblock%} dichiara un placeholder che potrà essere utilizzato in altri file html per riempire dinamicamente lo spazio che intercorre all'interno del *nome_block*. Invece attraverso la sintassi {{…}} possiamo utilizzare funzioni o variabili provenienti da *Python*, come ad esempio url_for, utile per indirizzare ad un file sorgente o link.

- body - dove oltre ad includere il corpo della nostra pagina e la sintassi *jinja2* per inserire placeholder, abbiamo i tag <header> e <footer> per definire lo stile e la struttura della parte alta e bassa del documento html.

Andiamo quindi a creare le nostre pagine html, attraverso i seguenti comandi da terminale:

```
$ nano app/templates/WebContent/homepage.html
```

```
$ nano app/templates/WebContent/register_page.html
```

```
$ nano app/templates/WebContent/login_page.html
```

```
$ nano app/templates/WebContent/welcome.html
```

```
{% extends "templatesFolder/project_template.html"%}
{% block header %}
{% endblock %}
{% block body %}
<h1 class="font-weight-light mt-4 text-white">Prova pratica</h1>
<p class="lead text-white-50">Progetto buildato con flask, nginx e mongoDB</p>
<br>
<form action="{{url_for('register')}}">
    <input name="register" id="id_register" type="submit" class="btn btn-dark btn-lg btn-block"
value="Register page">
</form>
<br>
<form action="{{url_for('login')}}">
    <input name="login" id="id_login" type="submit" class="btn btn-dark btn-lg btn-block" value="Login
page">
</form>
<br>
{% endblock %}
{% block footer %}
{% endblock %}
```

Jinja

app/templates/WebContent/homepage.html

HTML

La pagina ***homepage.html*** viene richiamata quando navighiamo tramite un browser all'indirizzo locale (http://localhost/ o http://127.0.0.1/ della nostra macchina /server) e ci consentirà di navigare verso le pagine di registrazione (***register_page.html***) o di login (***login_page.html***).

Carlo Lomello - 0124001651

Francesco Mabilia - 0124001910

Roberto Vecchio - 0124001871

Carlo Lomello - 0124001651 **Francesco Mabilia - 0124001910** **Roberto Vecchio - 0124001871**

1

```
{% extends "templatesFolder/project_template.html"%}
{% block body %}
<h1 class="font-weight-light mt-4 text-white">Registrazione</h1>

<form id="idformInsert" action="{{url_for('register_conf')}}">
   <div class="form-group">
      <!-- EMAIL -->
      <input type="text" name="username"
             id="id_username" placeholder="Enter username"
             class="form-control mt-5" aria-describedby=“emailHelp""><br>
      <!-- END EMAIL -->

      <!-- PASSWORD -->
      <input type="password" class="form-control" name="password"
             id="id_password" placeholder="Enter password"><br>
      <input type="password" class="form-control"
            name="password2" id="id_password2"
            placeholder="Confirm password"><br>
      <!-- END PASSWORD -->

      <!-- INFO -->
      <input type="text" class="form-control" name="nome" id="id_nome"
             placeholder="Enter name"><br>
      <input type="text" class="form-control" name="cognome"
             id="id_cognome" placeholder="Enter surname"><br>
      <input type="text" class="form-control" name="telefono" id="id_telefono"
             placeholder="Enter telephone number">
      <!-- END INFO -->

      <input class="btn btn-dark btn-lg btn-block mt-5" type="button"
             onclick="myFunction()" value="Registrati!">
   </div>
</form>

{% endblock %}
```

2

```
{% block other %}
   <script>
      function myFunction() {
         var username = document.getElementById("id_username").value;
         var password = document.getElementById("id_password").value;
         var password2 = document.getElementById("id_password2").value;
         var nome = document.getElementById("id_nome").value;
         var cognome = document.getElementById("id_cognome").value;
         var telefono = document.getElementById("id_telefono").value;

         if(username == null || username == ""
         || password == null || password == ""
         || password2 == null || password2 == ""
         || nome == null || nome == ""
         || cognome == null || cognome == ""
         || telefono == null || telefono == ""
         ){
            window.alert("tutti i campi sono obbliegatori!");
         }else if (password != password2){
            window.alert("le password non coincidono");
         }
         else {
            document.getElementById("idformInsert").submit();
         }
      }
   </script>
{% endblock %}
```

templates/templatesFolder/register_page.html

La pagina ***register_page.html*** ci consentirà di effettuare una registrazione di un utente attraverso una *form* il quale richiede di inserire alcuni dati.
L'effettiva registrazione avverrà attraverso la funzione *python* ***register_conf*** che inserirà il nostro utente nel *database* per poi reindirizzarci alla *homepage.*

Carlo Lomello - 0124001651

Francesco Mabilia - 0124001910

Roberto Vecchio - 0124001871

```
{% extends "templatesFolder/project_template.html"%}
{% block body %}

<h1 class="font-weight-light mt-4 text-white">Login Page</h1>

<form id="LogIn" action="{{url_for('logged_in')}}">
    <div class="form-group">
        <input type="text" name="username" class="form-control mt-5" id="id_username" placeholder="username"><br>
        <input type="password" name="password"  class="form-control" id="id_password"placeholder="password">
        <input type="button" onclick="login()" class="btn btn-dark btn-lg btn-block mt-5" value="LogIn">
    </div>
</form>

{% endblock %}
{% block other %}

<script>
    function login() {
    var username = document.getElementById("id_username").value;
    var password = document.getElementById("id_password").value;
    if(username == null || username == "" || password == null || password == ""){
        window.alert("tutti i campi sono obbligatori!");
    }else{
        document.getElementById("LogIn").submit();
    }
}
</script>

{% endblock %}
```

Jinja

app/templates/WebContent/login_page.html

HTML

La pagina ***login_page.html*** ci consentirà di effettuare un login di un utente precedentemente registrato, per poi utilizzare la funzione python ***logged_in*** tramite una *form* che, come precedentemente spiegato, ci reindirizzerà alla welcome.html trasportando con se le informazioni relative all'utente inserito nel nostro *database*.

Carlo Lomello - 0124001651

Francesco Mabilia - 0124001910

Roberto Vecchio - 0124001871

```
{% extends "templatesFolder/project_template.html"%}
{% block body %}

<div class="justify-content-center">
    <h1 class="font-weight-light mt-4 text-white">Welcome</h1>

    <!-- CARD -->
    <div class="card border-dark mb-3 mt-3" style="max-width: 18rem;">
        <div class="card-body">
            <h5 class="card-title">User Info</h5>
            <img class="mt-1 mb-2" src="{{ url_for('static', filename='assets/user.svg') }}" id="user_logo">
            <p class="card-text mt-3"> nome : {{ nome }} </p>
            <p class="card-text"> cognome : {{ cognome }} </p>
            <p class="card-text"> telefono : {{telefono}} </p>
        </div>
    </div>
</div>

{% endblock %}
```

Jinja

app/templates/WebContent/welcome.html

HTML

Una volta effettuato l’accesso, la pagina ***welcome.html*** ci mostrerà i dati dell’utente.

Carlo Lomello - 0124001651

Francesco Mabilia - 0124001910

Roberto Vecchio - 0124001871

Ora possiamo creare la folder ***static*** che andrà ad ospitare i nostri *contenuti statici, ad esempio: file* ***css***, ***js***, ***assets***, *etc*.

```
$ mkdir app/static
```

In **static**, creeremo a nostra volta altre due cartelle per organizzare il contenuto. Quindi andiamo a creare la folder ***assets*** che conterrà immagini ed altri contenuti multimediali e successivamente la folder ***css***, che conterrà i nostri fogli di stile:

```
$ mkdir app/static/assets
```

```
$ mkdir app/static/css
```

Il nostro progetto avrà un unico *file css* di default che andremo ad utilizzare per tutte le nostre *pagine html:*

```
$ nano app/static/templatesFolder/project_style.css
```

Carlo Lomello - 0124001651

Francesco Mabilia - 0124001910

Roberto Vecchio - 0124001871

Carlo Lomello - 0124001651
Francesco Mabilia - 0124001910
Roberto Vecchio - 0124001871

1

```css
html,
body {
  height: 100%;
}

#page-content {
  flex: 1 0 auto;
}

#sticky-footer {
  flex-shrink: unset;
}

body {
  background: #007bff;
  background: linear-gradient(to right, #0062E6, #33AEFF);
}

img {
  height: 10%;
  width: 10%;
  filter: invert(0.2);
}
```

2

```css
label {
  color: white;
}

p{
  color: white;
}

#user_logo{
  width: 25%;
  height: 25%;
}

.card {
  background: rgba(255, 252, 252, 0);
  border: 3px solid;
  margin: 0 auto;
  float: none;
}
```

app/static/css/project_style.css

Adesso importiamo le due immagini utili a mostrare una corretta UI.
Queste possono essere reperibile ***cliccando qui***.

Carlo Lomello - 0124001651

Francesco Mabilia - 0124001910

Roberto Vecchio - 0124001871

Per impostazione predefinita, *MongoDB* consente agli utenti di accedere senza credenziali e concede privilegi illimitati. In questo passaggio, proteggeremo il database creando un utente dedicato per accedervi.

Per fare ciò, avremo bisogno del nome utente e della password di root impostati nelle variabili di ambiente del file ***docker-compose.yml***

In generale, è meglio evitare di utilizzare l'account amministrativo di *root* quando si interagisce con il *database*. Invece, andremo a creare un utente dedicato per la tua applicazione *Flask.*

Carlo Lomello - 0124001651

Francesco Mabilia - 0124001910

Roberto Vecchio - 0124001871

Per creare un nuovo utente, avviare prima una shell interattiva sul contenitore mongodb:

```
$ docker exec -it mongodb bash
```

Una volta all'interno del contenitore, accedere all'account amministrativo principale di MongoDB:

```
# mongo -u user
```

Successivamente inseriremo la password richiesta:

```
# password
```

Creiamo un nuovo database attraverso il comando:

```
> use flaskdb
```

Carlo Lomello - 0124001651 **Francesco Mabilia - 0124001910** **Roberto Vecchio - 0124001871**

Questo comando crea un utente chiamato ***flaskuser*** con accesso *readWrite* al database ***flaskdb***.

Assicuriamoci di utilizzare una password sicura nel campo pwd.

User e *pwd* qui sono i valori definiti nel file ***docker-compose.yml*** nella sezione delle variabili di ambiente per il servizio di *Flask*.

```
> db.createUser({user: ‘flaskuser’, pwd: ‘password’, roles: [{role: 'readWrite', db: ‘flaskdb’}]})
```

Disconnettiamo l’utente root con il seguente comando:

```
> exit
```

Carlo Lomello - 0124001651 **Francesco Mabilia - 0124001910** **Roberto Vecchio - 0124001871**

Ora possiamo validificare l'utente appena creato attraverso il seguente comando:

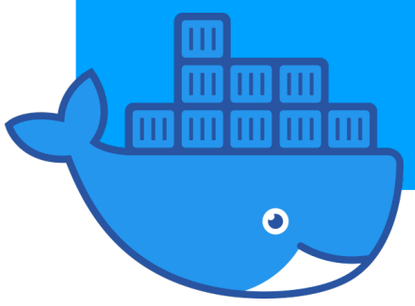
```
# mongo -u flaskuser -p password --authenticationDatabase flaskdb
```

Adesso non ci resta che uscire dal nostro utente *flaskuser* mediante il comando

```
> exit
```

Per verificare la correttezza del nostro lavoro eseguendo quindi i rispettivi container, andiamo a puntare da terminale al *root path* di progetto e lanciamo il comando:

```
$ docker-compose up --build
```

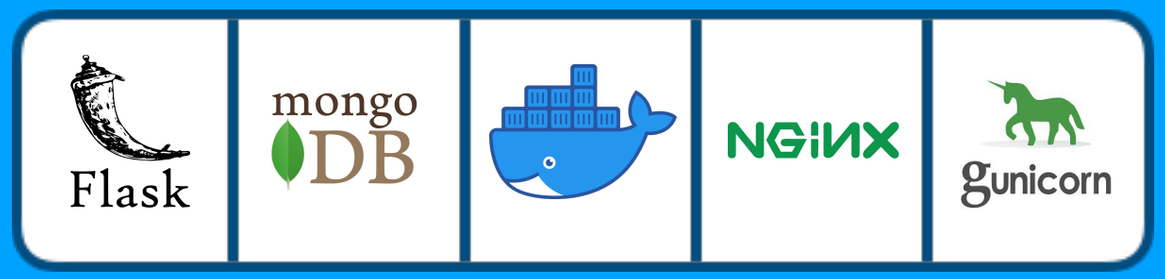


Carlo Lomello - 0124001651

Francesco Mabilia - 0124001910

Roberto Vecchio - 0124001871

Possiamo di seguito notare un esempio di utilizzo :



Carlo Lomello - 0124001651

Francesco Mabilia - 0124001910

Roberto Vecchio - 0124001871

Carlo Lomello - 0124001651 Francesco Mabilia - 0124001910 Roberto Vecchio - 0124001871

Prova pratica

Progetto buildato con flask, nginx e mongoDB

Register page

Login page

Carlo Lomello - 0124001651 Francesco Mabilia - 0124001910 Roberto Vecchio - 0124001871

Carlo Lomello - 0124001651 Francesco Mabilia - 0124001910 Roberto Vecchio - 0124001871

Registrazione

RossiMario

••••••••

••••••••

Mario

Rossi

3409865234

Registrati!

Carlo Lomello - 0124001651 Francesco Mabilia - 0124001910 Roberto Vecchio - 0124001871

Carlo Lomello - 0124001651 Francesco Mabilia - 0124001910 Roberto Vecchio - 0124001871

Login Page

RossiMario

••••••••

LogIn

Carlo Lomello - 0124001651 Francesco Mabilia - 0124001910 Roberto Vecchio - 0124001871

Carlo Lomello - 0124001651
Francesco Mabilia - 0124001910
Roberto Vecchio - 0124001871

Welcome

User Info

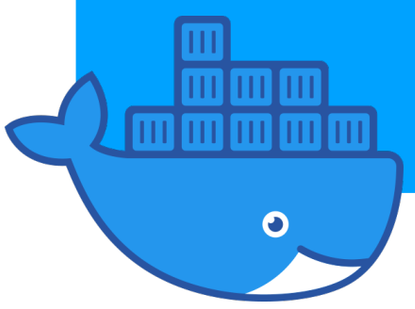
nome : Mario

cognome : Rossi

telefono : 3409865234

Carlo Lomello - 0124001651
Francesco Mabilia - 0124001910
Roberto Vecchio - 0124001871

Possiamo Scaricare il progetto pre-impostato ***cliccando qui***.

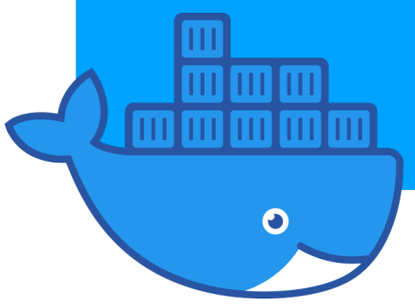


Carlo Lomello - 0124001651

Francesco Mabilia - 0124001910

Roberto Vecchio - 0124001871

È possibile aggiungere un *live reloader* al progetto per agevolare la fase di sviluppo *frontend side*, in modo da non dover rilanciare ripetitivamente il comando
docker-compose up --build
e quindi possiamo visualizzare velocemente ogni piccola modifica.



Carlo Lomello - 0124001651

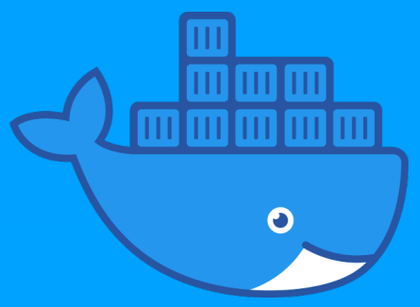
Francesco Mabilia - 0124001910

Roberto Vecchio - 0124001871

Carlo Lomello - 0124001651 **Francesco Mabilia - 0124001910** **Roberto Vecchio - 0124001871**

Ci basterà quindi aggiungere il seguente codice ai *services* nel file ***docker-compose.yml***

```
live-reloader:
  image: apogiatzis/livereloading
  container_name: livereloader
  privileged: true
  environment:
    - RELOAD_DELAY=0.5              # seconds
    - RELOAD_CONTAINER=flask
volumes:
  - "/var/run/docker.sock:/var/run/docker.sock"
  - ./app:/var/www
```



docker-compose.yml

Andiamo ad analizzare i seguenti elementi:

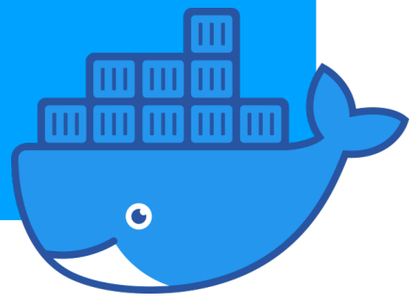
- **RELOAD_DELAY** specifica il tempo prima dell'esecuzione di un riavvio, pertanto più modifiche continue al file comportano un solo riavvio.

- **RELOAD_CONTAINER** è il nome del container che si vuole riavviare al momento della modifica dei file ad esso correlati.

- **./app:/var/www** specifica il volume di cui si vogliono monitorare i cambiamenti (infatti questo corrisponderà al volume del container specificato in reload_container)

Andiamo a puntare da terminale al *root path* di progetto e lanciamo il comando:

```
$ docker-compose up --build
```



Carlo Lomello - 0124001651

Francesco Mabilia - 0124001910

Roberto Vecchio - 0124001871

Carlo Lomello - 0124001651

Francesco Mabilia - 0124001910

Roberto Vecchio - 0124001871