Music Database Query App for Distributed Systems on Cloud

Lorenzo Maria Alberto Paoria

22 luglio 2025

Sommario

Il progetto "Music Database Query App for Distributed Systems on Cloud" rappresenta un'applicazione distribuita per la gestione di query su database musicali implementata su infrastruttura cloud AWS.

1 L'idea

L'idea del progetto nasce nell'ambito del corso di Sistemi Cloud e Laboratorio. Avendo già sviluppato un'applicazione per il progetto del corso di Sistemi Distribuiti e Laboratorio, ho deciso di riutilizzarla come base. La sfida principale consiste nel progettare un'architettura cloud in grado di gestire carichi di lavoro variabili, garantire la persistenza dei dati e mantenere prestazioni ottimali anche in presenza di picchi di traffico.

La soluzione permette a più client di connettersi simultaneamente al server, eseguire query sul database musicale e ricevere risultati in tempo reale, il tutto gestito attraverso una CLI.

Figura 1: Logo

2 Tecnologie utilizzate

2.1 Amazon Web Services (AWS)

AWS fornisce l'infrastruttura cloud per l'hosting dell'applicazione. I servizi utilizzati includono:

- **EC2**: Istanza virtuale per l'hosting del server applicativo
- **RDS**: Database PostgreSQL per l'uso applicativo
- **VPC**: Rete virtuale privata per la sicurezza
- **Security Groups**: Firewall
- **SNS**: Servizio di notifiche per il monitoraggio
- **SQS**: Code di messaggi per il logging
- **NLB**: Network Load Balancer per la distribuzione del carico

2.2 Java

Linguaggio di programmazione principale per lo sviluppo dell'applicazione. Java è stato scelto per:

- Portabilità multi-piattaforma
- Robustezza nella gestione delle eccezioni
- Ricco ecosistema di librerie per database e networking
- Supporto nativo per la programmazione multi-threaded

2.3 PostgreSQL

Sistema di gestione database relazionale scelto per:

- Affidabilità e conformità ACID
- Supporto per transazioni complesse
- Ottimizzazione per carichi di lavoro misti (lettura/scrittura)
- Integrazione nativa con Amazon RDS

2.4 Docker

Containerizzazione dell'applicazione per:

- Deployment consistente tra ambienti diversi
- Isolamento delle dipendenze
- Scalabilità orizzontale
- Facilità di gestione e aggiornamento

2.5 Python

Utilizzato per gli script di automazione dell'infrastruttura:

- Deploy automatizzato delle risorse AWS
- Aggiornamento dei secrets GitHub (indirizzo ip server e chiave)
- Aggiornamento degli indirizzi ip, dns e rds endopoint
- Monitoraggio della coda di messaggi

2.6 GitHub

Utilizzato come sostegno in quanto:

- Sistema di versionamento del codice
- Automazione dei processi di CI/CD tramite Actions

3 Architettura del Sistema

3.1 Architettura Generale

Il sistema adotta un'architettura client-server distribuita su cloud con i seguenti componenti principali:

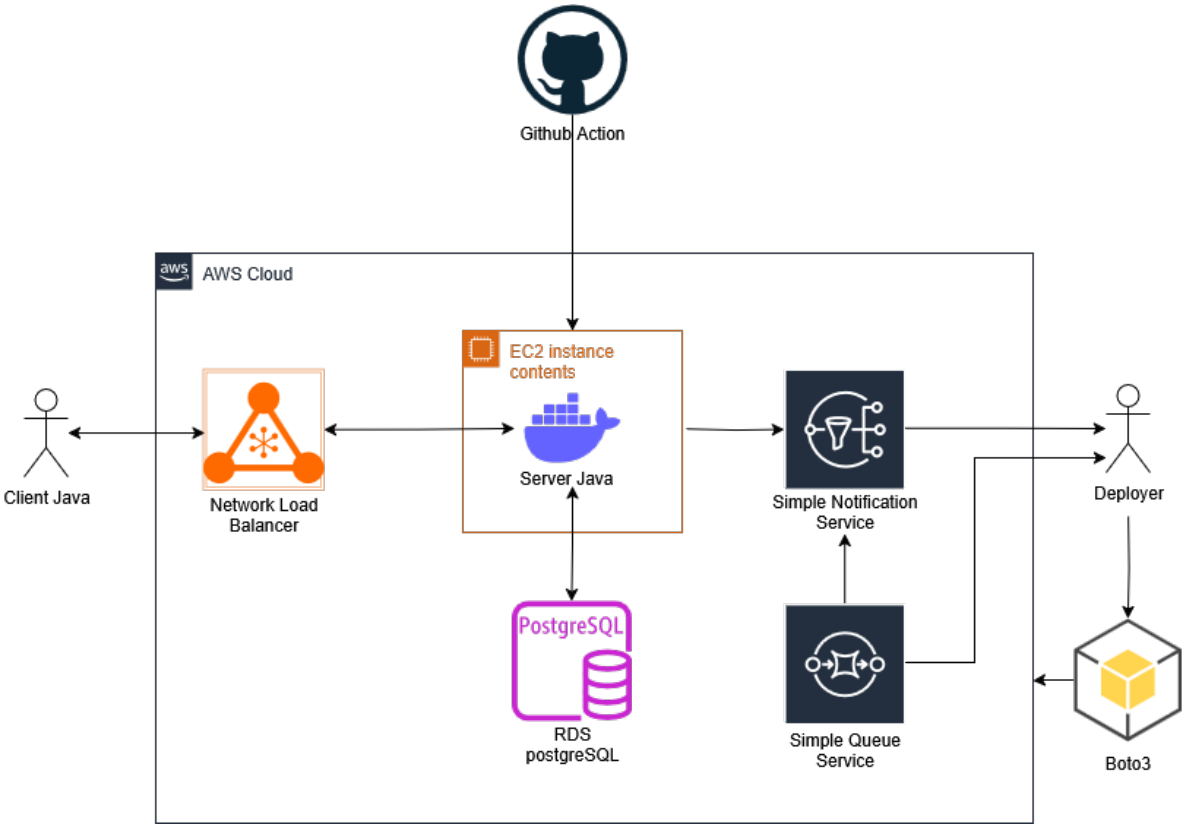

Figura 2: Diagramma dell'architettura generale del sistema

- **Client Java**: Interfaccia CLI Java per l'interazione utente
- **Deployer**: Amministratore del l'intero sistema
- **Server Java**: Server Java multi-threaded ospitato su container Docker runnato su EC2
- **RDS postgreSQL**: PostgreSQL database gestito tramite Amazon RDS
- **Network Load Balancer**: Load Balancer per la distribuzione del traffico che usa TCP
- **Simple Notification Service**: SNS per notificare lo stato del server
- **Simple Queue Service**: SQS per il logging
- **GitHub Actions**: per il CI/CD

3.2 Struttura del Database

Il Database è progettato per la memorizzazione e gestione dei dati relativi ad una piattaforma di streaming musicale.

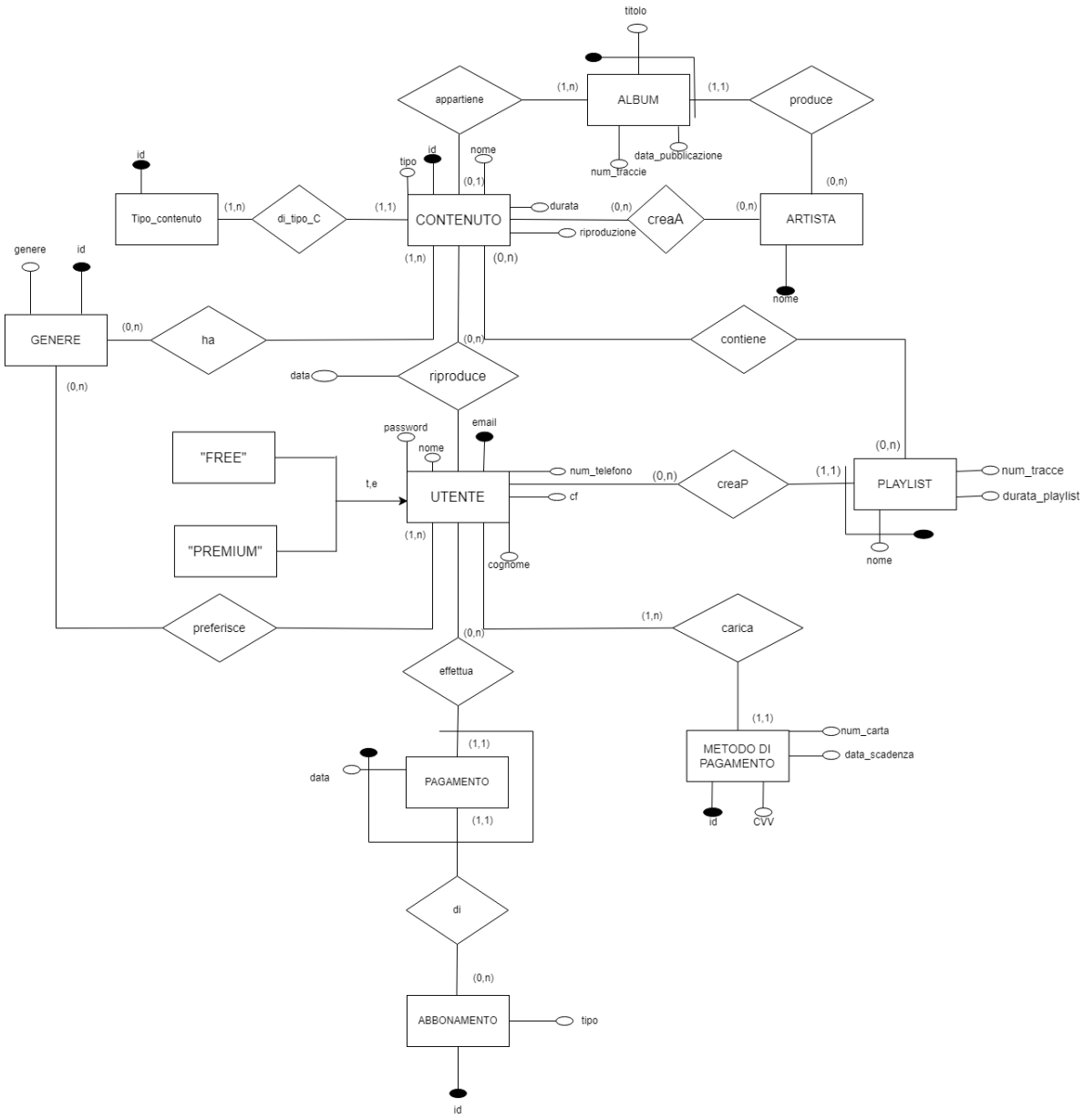

Figura 3: Schema ER del database musicale

4 Panoramica

La cartella 'scripts/infrastructure' contiene gli script Python per l'automazione del deploy e gestione dell'infrastruttura AWS. Ogni script ha un ruolo specifico.

4.1 Panoramica dei File Infrastructure

- **deploy_music_app.py**: Script principale per il deploy completo dell'infrastruttura
- **setup_nlb.py**: Configurazione del Network Load Balancer che è opzionale per il funzionamento base dell'applicativo
- **update_github_secrets.py**: Aggiornare i secrets GitHub per il corretto funzionamento della Action
- **update_java_config_on_ec2.py**: Aggiornamento configurazione Java in locale e push della repo
- **monitor_sqs.py**: Monitoraggio in tempo reale della coda di messaggi
- **user_data_script.sh**: Script di inizializzazione per l'istanza EC2
- **deploy_config.json**: File con le specifiche di tutti i componenti creati generato automaticamente alla fine del deploy dell'infrastruttura
- **my-ec2-key.pem**: Chiave privata per accesso SSH alle istanze EC2

5 Script

Il sistema utilizza script Python per l'automazione del deploy su AWS:

5.1 deploy_music_app.py - Script Principale

Listing 1: Struttura principale dello script di deploy

```python
def main():

    # Pulizia delle risorse AWS
    if "--clean" in os.sys.argv:
        delete_resources(ec2, rds, KEY_PAIR_NAME,
                         DB_INSTANCE_IDENTIFIER,
                         'MusicAppRDSSecurityGroup',
                         'MusicAppEC2SecurityGroup',
                         skip_rds)
    return

    # Deploy EC2 e RDS
    ec2_client = boto3.client('ec2', region_name=REGION)
    rds_client = boto3.client('rds', region_name=REGION)

    # Step 1: Gestione credenziali AWS
    aws_credentials = read_aws_credentials()

    # Step 2: Creazione Key Pair EC2
    key_pair_name = get_key_pair(ec2_client, KEY_PAIR_NAME)

    # Step 3: Configurazione VPC e Security Groups
    vpc_id, rds_sg_id, ec2_sg_id = create_vpc_and_security_groups(
        ec2_client, rds_client)

    # Step 4-5: Deploy RDS e inizializzazione database
    rds_endpoint = deploy_rds_instance(rds_client, rds_sg_id)
    initialize_database(rds_endpoint, schema_sql, data_sql)

    # Step 6-7: Setup notifiche SNS e code SQS
    topic_arn = setup_sns_notification(REGION, topic_name, email)
    queue_url = setup_sqs_logging_queue(REGION, queue_name, topic_arn)

    # Step 8: Deploy istanze EC2
    deploy_ec2_instances(ec2_client, ec2_sg_id, user_data_script)

    # Step 9: Salvataggio informazioni nel deploy_config.json
    config = {
            "server_public_ip": server_public_ip,
            "server_private_ip": server_private_ip,
            "rds_endpoint": rds_endpoint,
            "db_username": DB_MASTER_USERNAME,
            "db_password": DB_MASTER_PASSWORD,
            "db_name": DB_NAME,
            "key_pair_name": key_pair_name_actual
    }
```

Caratteristiche principali:

- **Idempotenza**: L'operazione può essere eseguita più volte senza causare problemi, in quanto, se le risorse da creare sono già presenti, vengono semplicemente riutilizzate.
- **Rollback**: Supporta una pulizia completa o parziale delle risorse create. Utilizzando gli argomenti `--clean --nords`, è possibile eliminare tutte le risorse generate, mantenendo intatto il database (la cui creazione richiede tempi lunghi). Oppure è possibile effetturare una pulizia completa tramite l'argomento `--clean`
- **Logging**: Fornisce un output dettagliato per ogni operazione, organizzato in sezioni come `[SECTION]`, `[INFO]`, `[SUCCESS]`, `[WARNING]` e `[ERROR]`.
- **Gestione degli errori**: Implementa una gestione robusta degli errori legati ai servizi AWS.

5.2 setup_nlb.py - Network Load Balancer

Script dedicato alla configurazione del NLB:

Listing 2: Configurazione Network Load Balancer

```
def main():

        # Pulizia risorse create
        if "--clean" in os.sys.argv:
            cleanup_nlb_resources()
        return

        # lettura della configurazione di deploy esistente
        deploy_config = read_deploy_config()

        # inizializzazione dei client AWS
        ec2_client = boto3.client('ec2', region_name=REGION)
        elbv2_client = boto3.client('elbv2', region_name=REGION)

        # STEP 1: recupero delle informazioni di rete
        vpc_id, subnet_ids = get_default_vpc_and_subnets(ec2_client)

        # STEP 2: identificazione dell'istanza server
        server_instance_id = get_server_instance_id(ec2_client)

        # STEP 3: creazione del target group
        target_group_arn = create_target_group(elbv2_client, vpc_id)

        # STEP 4: creazione del Network Load Balancer
        nlb_arn, nlb_dns = create_nlb(elbv2_client, subnet_ids)

        # STEP 5: configurazione del listener
        listener_arn = create_listener(elbv2_client, nlb_arn, target_group_arn)

        # STEP 6: registrazione dell'istanza nel target group
        register_target(elbv2_client, target_group_arn, server_instance_id)

        # STEP 7: aggiornamento della configurazione di deploy
        update_deploy_config(nlb_dns, NLB_PORT)
```

Caratteristiche principali:

- **Idempotenza**: L'operazione può essere eseguita più volte senza causare problemi.
- **Rollback**: Supporta una pulizia completa delle risorse create attraverso l'argomento `--clean`
- **Logging**: Fornisce un output dettagliato per ogni operazione anche esso suddiviso in sezioni.
- **Gestione degli errori**: Implementa una gestione robusta degli errori legati ai servizi AWS.
- **Indirizzamento**: Distribuzione automatica del traffico.
- **Health Check**: Monitoraggio ogni 30 secondi.
- **Scalabilità**: Supporto per multiple istanze.
- **Performance**: Latenza ultra-bassa (Layer 4) in quanto lavora tramite TCP.

5.3 update_github_secrets.py - Integrazione CI/CD

Script per aggiorna i GitHub Secrets utili per il funzionamento delle Actions:

Listing 3: Aggiornamento GitHub Secrets

```
def main():

    script_dir = os.path.dirname(os.path.abspath(__file__))

    # STEP 1: caricamento del token GitHub dall'ambiente
    github_token = load_environment()

    # STEP 2: recupero delle informazioni del repository Git
    owner, repo = get_repo_info()

    # STEP 3: verifica dell'esistenza dei file necessari
    config_file = os.path.join(script_dir, "deploy_config.json")
    pem_file = os.path.join(script_dir, "my-ec2-key.pem")

    # STEP 4: lettura delle configurazioni e della chiave privata
    with open(config_file, 'r') as f:
        config = json.load(f)

    with open(pem_file, 'r') as f:
        pem_content = f.read()

    # STEP 5: recupero della chiave pubblica del repository
    public_key_info = get_public_key(owner, repo, github_token)

    # STEP 6: aggiornamento del secret EC2_HOST
    print(f"[STEP] Aggiornamento del secret EC2_HOST con valore: {config['
        server_public_ip']}")
    if not update_secret(owner, repo, github_token, "EC2_HOST",
                         config['server_public_ip'],
                         public_key_info['key_id'],
                         public_key_info['key']):
        success = False

    # STEP 7: aggiornamento del secret EC2_SSH_KEY
    print("[STEP] Aggiornamento del secret EC2_SSH_KEY in corso...")
    if not update_secret(owner, repo, github_token, "EC2_SSH_KEY",
                         pem_content,
                         public_key_info['key_id'],
                         public_key_info['key']):
        success = False

    # STEP 8: verifica del risultato finale
    if success:
        print("[SUCCESS] Tutti i secrets sono stati aggiornati correttamente!")
        print("[INFO] Ora puoi procedere con il push per triggerare il deploy!")
    else:
        print("[ERROR] Alcuni secrets non sono stati aggiornati correttamente.")
```

Caratteristiche principali:

- **Crittografia**: Secrets crittografati con chiave pubblica del repo
- **API GitHub**: Utilizzo sicuro delle API GitHub
- **Validazione**: Controlli di integrità sui dati

5.4 update_java_config_on_ec2.py - Aggiornamento Remoto

Script per aggiornare la configurazione Java in locale e fare una push alla repository GitHub:

Listing 4: Aggiornamento configurazione via SSH

```
def main():

    # lettura della configurazione di deploy
    with open("deploy_config.json", "r") as f:
        config = json.load(f)

    # STEP 1: determinazione della configurazione client (NLB/EC2)
    nlb_enabled = config.get("nlb_enabled", False)
    if nlb_enabled and "nlb_dns" in config and "nlb_port" in config:
        # configurazione NLB
        CLIENT_TARGET_HOST = config["nlb_dns"]
        CLIENT_TARGET_PORT = str(config["nlb_port"])
        print(f"[INFO] Configurazione client per Network Load Balancer: {
            CLIENT_TARGET_HOST}:{CLIENT_TARGET_PORT}")
    else:
        # configurazione EC2
        CLIENT_TARGET_HOST = SERVER_EC2_PUBLIC_IP
        CLIENT_TARGET_PORT = SERVER_APPLICATION_PORT
        print(f"[INFO] Configurazione client per connessione diretta EC2: {
            CLIENT_TARGET_HOST}:{CLIENT_TARGET_PORT}")

    print("[STEP] Avvio del processo di aggiornamento della configurazione...")

    # STEP 2: costruzione dei percorsi ai file di configurazione
    script_dir = os.path.dirname(os.path.abspath(__file__))
    project_root = os.path.abspath(os.path.join(script_dir, "..", ".."))

    server_config_path = os.path.join(project_root, "mvnProject-Server", "src", "main"
        , "java", "com", "example", "config", "DatabaseConfig.java")
    server_db_properties_path = os.path.join(project_root, "mvnProject-Server", "src",
         "main", "java", "com", "example", "config", "database.properties")
    client_config_path = os.path.join(project_root, "mvnProject-Client", "src", "main"
        , "java", "com", "example", "DatabaseClient.java")

    # STEP 3: aggiornamento dei file di configurazione Java
    update_local_java_config(
        server_ip=SERVER_EC2_PRIVATE_IP,
        server_port=SERVER_APPLICATION_PORT,
        rds_endpoint=RDS_ENDPOINT,
        db_username=DB_USERNAME,
        db_password=DB_PASSWORD,
        client_server_ip=CLIENT_TARGET_HOST,
        client_server_port=CLIENT_TARGET_PORT,
        server_config_path=server_config_path,
        server_db_properties_path=server_db_properties_path,
        client_config_path=client_config_path
    )

    # STEP 4: commit e push delle modifiche per triggerare GitHub Actions
    git_commit_and_push()
```

Caratteristiche principali:

- **Auto-detection**: Rileva automaticamente presenza NLB tramite deploy_config.json
- **Configuration Management**: Aggiorna configurazioni Java in locale
- **Git Integration**: Commit e push automatici che avvieranno la GitHub Action

5.5 monitor_sqs.py - Monitoraggio Real-time

Sistema di monitoraggio per log distribuiti:

Listing 5: Monitoraggio SQS in tempo reale

```python
def monitor_queue():
    # inizializzazione del client SQS
    sqs_client = boto3.client('sqs', region_name=REGION)

    # recupero dell'URL della coda
    queue_url = sqs_client.get_queue_url(QueueName=QUEUE_NAME)['QueueUrl']

    # visualizzazione delle informazioni di avvio
    print(f"[INFO] Monitoraggio coda SNS avviato")
    print(f"[INFO] Coda: {QUEUE_NAME}")
    print(f"[INFO] In ascolto di nuovi messaggi...")
    print("-" * 60)

    message_count = 0

    # loop principale di monitoraggio
    while True:
        try:
            # lettura e visualizzazione dei nuovi messaggi
            new_message_count = read_and_display_messages(sqs_client, queue_url)
            message_count += new_message_count

            # indicatore di attesa se non ci sono nuovi messaggi
            if new_message_count == 0:
                current_time = datetime.datetime.now().strftime('%H:%M:%S')
                print(f"[WAIT] {current_time} - In ascolto... (Tot: {message_count})",
                      end='\r')

        except KeyboardInterrupt:
            # gestione dell'interruzione da tastiera
            print(f"\n\n[SUCCESS] Monitoraggio terminato. Messaggi processati: {
                message_count}")
            break
        except Exception as e:
            # gestione degli errori con retry automatico
            print(f"\n[ERROR] Errore durante il monitoraggio: {e}")
            time.sleep(5)

def main():
    monitor_queue()
```

Caratteristiche principali:

- **Real-time**: Visualizzazione immediata dei nuovi messaggi
- **Timezone**: Conversione automatica fuso orario

5.6 user_data_script.sh - Inizializzazione EC2

Script bash per l'inizializzazione automatica delle istanze EC2:

Listing 6: Script di inizializzazione EC2

```
#!/bin/bash
sudo dnf update -y
# installa Git, Docker e AWS CLI
sudo dnf install -y  git docker awscli
sudo systemctl enable docker
sudo systemctl start docker
sudo usermod -aG docker ec2-user
newgrp docker

# configura AWS CLI per ec2-user
sudo -u ec2-user mkdir -p /home/ec2-user/.aws
sudo -u ec2-user cat <<EOF > /home/ec2-user/.aws/credentials
[default]
aws_access_key_id=AWS_ACCESS_KEY_ID_PLACEHOLDER
aws_secret_access_key=AWS_SECRET_ACCESS_KEY_PLACEHOLDER
aws_session_token=AWS_SESSION_TOKEN_PLACEHOLDER
EOF


# clono il repository GitHub e imposto i permessi
git clone https://github.com/lorenzopaoria/repo.git $APP_DIR
sudo chown -R ec2-user:ec2-user $APP_DIR

# verifico che il Dockerfile esista nel repository
if [ -f "$APP_DIR/Dockerfile.dockerfile" ]; then
    # rinomino il Dockerfile esistente per Docker
    mv "$APP_DIR/Dockerfile.dockerfile" "$APP_DIR/Dockerfile"
    echo "Using existing Dockerfile from repository"
else
    echo "Warning: Dockerfile.dockerfile not found in repository"
    exit 1
fi

# build della Docker image
echo "Building the Docker image..."
cd $APP_DIR
docker build -t music-server-app .

# run del container Docker
echo "Running the Docker container..."
docker run -d -p 8080:8080 --name musicapp-server music-server-app

# notifica SNS di completamento setup del server (eseguito come ec2-user)
sudo -u ec2-user aws sns publish --region us-east-1 --topic-arn $(message)
```

Caratteristiche principali:

- **Zero Configuration**: Setup completamente automatico
- **Docker Integration**: Containerizzazione del'app server con porta 8080 esposta
- **AWS Integration**: Configurazione automatica AWS CLI
- **Notification**: Notifica SNS al completamento

5.7 Dockerfile.dockerfile - File configurazione del container

Listing 7: dockerfile per container

```
# uso di un'immagine ufficiale di Maven con Java 17 come immagine di base
FROM maven:3.9-eclipse-temurin-17

# imposta la directory di lavoro all'interno del contenitore
WORKDIR /app

# copia l'intero progetto all'interno del contenitore
COPY . .

# Questo compilera' il codice e scarichera' le dipendenze
RUN mvn -f mvnProject-Server/pom.xml clean install

# Imposta la directory di lavoro sulla cartella del progetto server per il comando
    successivo
WORKDIR /app/mvnProject-Server

# espone la porta 8080 per consentire il traffico all'applicazione
EXPOSE 8080

# Questo e' il comando che verra' eseguito all'avvio del contenitore
CMD ["mvn", "-Pserver", "exec:java"]
```

Caratteristiche principali:

- **Immagine Base Ufficiale**: Usa l'immagine Docker ufficiale maven:3.9-eclipse-temurin-17 con Java 17 preinstallato
- **Build Automatizzato**: Esegue automaticamente mvn clean install per compilare il progetto e risolvere le dipendenze
- **Directory Setup**: Imposta directory di lavoro chiare per build e runtime
- **Port Mapping**: Espone la porta `8080` per accesso esterno al server
- **Avvio del Server**: Esegue il server Java via Maven usando il profilo `server` all'avvio del container

5.8 Deploy Docker Container on EC2 - GitHub Actions Workflow

Workflow YAML per il deploy automatico dell'applicazione su un'istanza EC2 usando Docker e GitHub Actions.

Listing 8: Workflow GitHub Actions per il deploy EC2 con Docker

```yaml
name: Deploy Docker Container on EC2

on:
  push:
    branches: [ main, master ]

jobs:
  deploy:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout code
        uses: actions/checkout@v4

      - name: Validate EC2 connection
        uses: appleboy/ssh-action@v1.0.3
        with:
          host: ${{ secrets.EC2_HOST }}
          username: ec2-user
          key: ${{ secrets.EC2_SSH_KEY }}
          timeout: 60s
          script: |
            echo "Connessione EC2 stabilita con successo"
            whoami
            docker --version || echo "Docker non installato"

      - name: Deploy to EC2
        uses: appleboy/ssh-action@v1.0.3
        with:
          host: ${{ secrets.EC2_HOST }}
          username: ec2-user
          key: ${{ secrets.EC2_SSH_KEY }}
          timeout: 600s
          command_timeout: 30m
          script: |
            set -e

            cd /home/ec2-user/repo

            git fetch origin
            git reset --hard origin/main

            docker ps -q --filter "publish=8080" | xargs -r docker stop || true
            docker ps -aq --filter "publish=8080" | xargs -r docker rm || true

            sleep 5

            docker system prune -f

            echo "Ricostruendo il container Docker..."
            docker build -f Dockerfile.dockerfile -t musicapp-server

            docker run -d --name musicapp-server -p 8080:8080 --restart unless-stopped
                 musicapp-server

            sudo -u ec2-user aws sns publish --region us-east-1 --topic-arn $(message)

            echo "Deploy completato!"
```

Caratteristiche principali:

- **CI/CD Automatica**: Deploy automatico su EC2 ad ogni push su `main`
- **Connessione Sicura**: Connessione SSH al server EC2 tramite GitHub Secrets
- **Repo Sync**: Sincronizzazione forzata con `git fetch` e `git reset --hard`

- **Gestione Porta 8080**: Terminazione forzata di processi attivi e verifica disponibilità
- **Pulizia Docker**: Cleanup completo di container e immagini inutilizzate
- **Build & Run**: Costruzione della Docker image e avvio container con policy di riavvio
- **Health Check**: Verifica che il container sia attivo e che l'app risponda sulla porta `8080`
- **Notifica SNS**: Invio notifica al topic SNS AWS al termine del deploy

6 Workflow Operativo del Sistema

6.1 Processo di Deploy Completo

Il sistema implementa un workflow di deploy completamente automatizzato che segue questa sequenza:

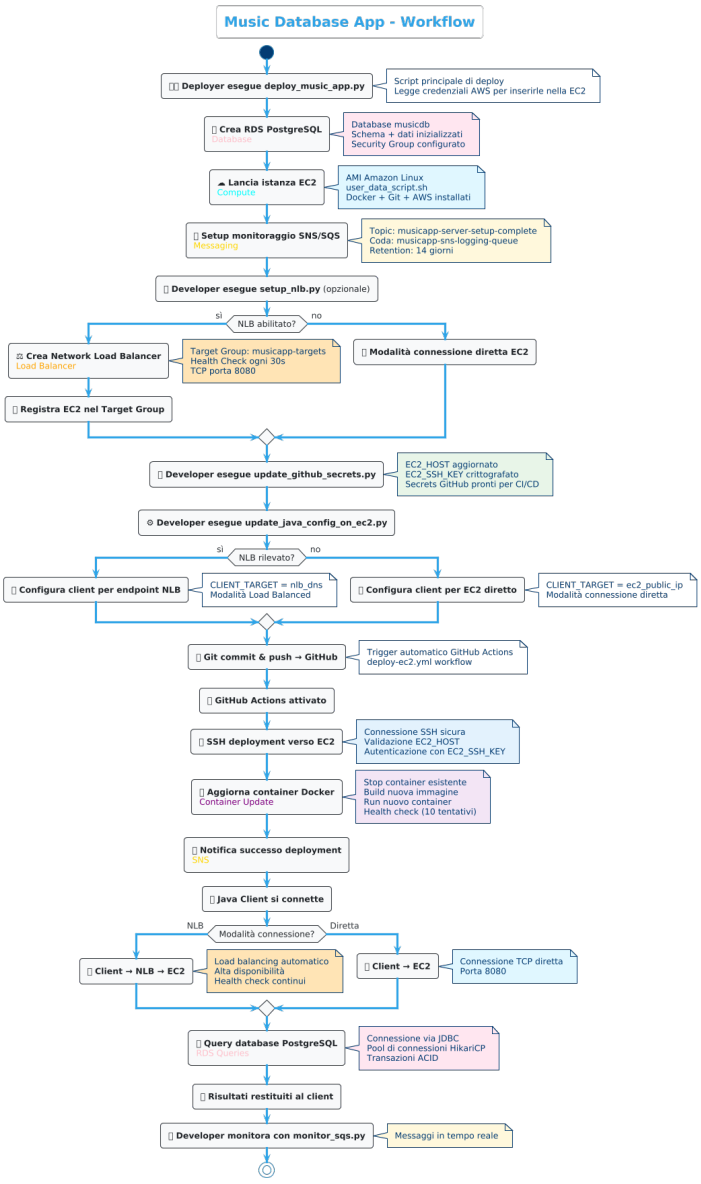

Figura 4: Diagramma del workflow

1. **Inizializzazione Infrastruttura**

```
python deploy_music_app.py
```

2. **Configurazione Load Balancer (Opzionale)**

```
python setup_nlb.py
```

3. **Sincronizzazione GitHub Secrets**

```
python update_github_secrets.py
```

4. **Aggiornamento Configurazione**

```
python update_java_config_on_ec2.py
```

5. **Monitoraggio Sistema**

```
python monitor_sqs.py
```

6.2 Sistema di Pulizia Risorse

Il sistema include opzioni flessibili per la pulizia delle risorse:

Pulizia completa:

```
python deploy_music_app.py --clean
```

- Elimina tutte le risorse AWS (EC2, RDS, NLB, SNS, SQS)
- Tempo stimato: 15-20 minuti (causa eliminazione RDS)
- Per documentazione dopo aver eliminato SQS si deve aspettare 1 minuto per poterlo ricreare

Pulizia parziale (mantiene database):

```
python deploy_music_app.py --clean --nords
```

- Elimina EC2, NLB, SNS, SQS ma mantiene RDS
- Tempo stimato: 2-3 minuti

Pulizia solo NLB:

```
python setup_nlb.py --clean
```

- Elimina solo Network Load Balancer
- Mantiene tutto il resto attivo

6.3 Prerequisiti e Configurazione

Il sistema richiede la seguente configurazione:

Listing 9: Configurazione credenziali AWS

```
# File ~/.aws/credentials
[default]
aws_access_key_id = YOUR_ACCESS_KEY_ID
aws_secret_access_key = YOUR_SECRET_ACCESS_KEY
aws_session_token = YOUR_SESSION_TOKEN  # opzionale per credenziali temporanee come
    Learner Lab
```

Listing 10: Configurazione GitHub per CI/CD

```
# File .env nella root del progetto
GITHUB_TOKEN=your_github_token
GITHUB_TOKEN_API=your_api_token
REPO=username/repository-name
SECRET_NAME=deployment_secrets
PEM_PATH=path/to/private/key
```

6.4 Tempi di Deployment

- **Deploy completo da zero**: 15-20 minuti
- **Setup Network Load Balancer**: 3-5 minuti
- **Aggiornamento configurazione**: 1-2 minuti
- **Pulizia completa risorse**: 15-20 minuti (incluso RDS)
- **Pulizia parziale (no RDS)**: 2-3 minuti