Deception Component Generator - Database

Riccardo Barbieri

14 giugno 2024

Indice

1 Introduzione

In questo documento verrà discussa l'implementazione di un componente aggiuntivo generabile tramite `deception-kit`. La nuova versione del toolkit supporta la generazione di un *database relazionale*.

La generazione di questo componente supporta le seguenti funzionalità:

- generazione di schemi da zero
- popolamento automatico con dati realistici
- creazione di utenti e assgnazione di permessi

2 Database

La prima scelta progettuale effettuata è stata la decisione di quale particolare DBMS utilizzare per la generazione del componente, è stato necessario scegliere un DBMS con distribuzioni sotto forma di immagine OCI e che permettesse di eseguire script di inizializzazione al primo startup per importare i dati generati.

Questa scelta si è successivamente rivelata superflua in quanto tutti i principali DBMS dispongono di definizioni di immagine OCI e supportano inizializzazione tramite script, si è deciso però di supportare PostgreSQL in quanto si tratta della soluzione open source che offre la maggiore quantità di feature avanzate.

Si denota che è comunque possibile istanziare le risorse generate in un server MySQL in quanto i due DBMS condividono tutte le funzionalità di SQL esteso che vengono usate dal generatore così come il meccanismo di inizializzazione che verrà dettagliato in seguito.

2.1 Definizione componente

La definizione delle caratteristiche del componente da generare è costituita da una estensione della specifica deception-def usata anche per il componente `idprovider`.

La specifica per il componente `database` richiede la definizione delle seguenti proprietà:

- `connection`: definisce informazioni per la connesione al server
- `databases`: dichiara i nomi dei database da creare
- `tables`: specifica le caratteristiche delle tabelle da generare (dettagli in seguito)
- `users`: lista di utenti e dichiarazione di un utente admin principale

2.1.1 Tabelle

Per definire le tabelle da generare e popolare con dati fittizi si è deciso di permettere all'utente di specificare il nome della tabella e il database di appartenenza.

Per creare lo schema per una tabella, l'utente deve definire un `prompt`: questo testo viene usato dal generatore per generare una serie di nomi ci campi e relativi tipi sfruttando un large language model.

L'utente è inoltre in grado di specificare una serie di metaparametri che impongono al generatore limiti minimi e massimi sul numero di colonne e righe da generare per le tabelle.

```yaml
tables:
  min_columns_per_table: 5
  max_columns_per_table: 10
  min_rows_per_table: 5
  max_rows_per_table: 10
  definitions:
    - name: "flight_schedule"
      prompt: "Flight Schedule"
      database: "flight_db"
    - name: "passengers"
      prompt: "Flight Passengers"
      database: "customer_db"
```

2.1.2 Utenti

Gli utenti vengono definiti specificando uno username, una password e una serie di database ai quali hanno accesso dove vengono inclusi anche i permessi che possiedono su un determinato database.

```yaml
users:
  definitions:
    - username: user1
      password: password1
      databases:
        - name: flight_db
          permissions:
            - select
            - insert
            - update
```

2.2 Init script

Le risorse generate a partire dalla `deception-def` del componente vengono propriamente salvate in appositi file `.sql`, questi file vengono importati all'interno del con-

tainer alla sua inizializzazione e successivamente eseguiti dal runtime di PostgreSQL per creare le entità nel server.

Il meccanismo di inizializzazione implementato da PostgreSQL (e MySQL) esegue tutti gli script `.sql` che trova nella cartella `/docker-entrypoint-initdb.d/` del container con l'identità dell'utente principale specificato nella definizione del componente. È importante specificare che gli script di inizializzazione vengono eseguiti solo se non esistono già dati nel database per non entrare in conflitto con dati esistenti.

Questo meccanismo di inizializzione è molto flessibile in quanto permette anche di eseguire script `bash` complessi per configurare in modo più avanzato il server, questo particolare permetterà in futuro di aggiungere ulteriori configurazioni alla definizione del componente.

3 Generazione

3.1 Ristrutturazione endpoint API

L'API esposta dal componenente `deception-core` è stata estesa per supportare nuovi endpoint che implementano le strategie di generazione dei dati richiesti per il componente `database`.

Si è deciso di modificare la struttura degli endpoint dell'api per marcare meglio la separazione tra i componenti:

- gli endpoint nella forma `/generation/<component>/*` generano risorse statiche
- gli endpoint nella forma `/registration/<component>/*` generano risorse dinamiche sfruttando un runtime (es. Keycloak per idprovider)

3.2 Generazione componenti database

Al contrario del componente `idprovider`, il `database` non necessita di una fase di *registrazione* a runtime di entità, vengono infatti generate soltanto risorse statiche.

L'API è stata progettata per fornire le risorse al client (`deception-cli`) che la utilizza non sotto forma di file script già compilato ma in una apposita struttura dati intermedia dalla quale il client genera i file necessari, questa decisione è stata presa per non legare la generazione dei componenti a una tecnica specifica: si lascia la decisione di come strutturare l'output al client.

3.3 Tabelle

La generazione delle tabelle avviene sfruttando lo stesso servizio utilizzato per i dati dell'`idprovider`, Mockaroo offre una feature che permette di generare una tabella SQL a partire da un prompt testuale sfruttando un modello di intelligenza artificiale.

La generazione si suddivide in una serie di passi:

- richiesta di generazione di una lista di tipi Mockaroo basata sul prompt fornito
- richiesta di generazione di una istruzione SQL `CREATE TABLE` dai tipi ottenuti
- richieste di generazione di una serie di istruzioni SQL `INSERT` con i dati fittizi

I numeri di colonne e record sono generati casualmente nell'intervallo fornito alla definizione del componente.

3.4 Utenti e database

La generazione delle istruzioni SQL per creare utenti e database viene gestita da appositi endpoint che compongono i comandi `CREATE DATABASE`, `CREATE USER` e `GRANT` a partire dalle definizioni trovate nella `deception-def` fornita.

4 Miglioramenti CLI

Le modifiche apportate all'interfaccia da linea di comando consistono in:

- ping a `deception-core` per verificare l'esecuzione del server
- generazione di componenti multipli
- sistema di configurazione

4.1 Generazioni multiple

La CLI è stata adattata per poter accettare configurazioni multiple con un solo comando.

Data la possibilità di generare più componenti viene aggiunto al nome del Dockerfile il nome del componente generato e sono suddivisi insieme agli asset necessari in apposite cartelle separate.

```
deception-cli generate -c "database" -d id.yaml -c "idprovider" -d db.yaml
```

4.2 Configurazione

È stato aggiunto un file di configurazione all'installazione della CLI che permette di specificare proprietà di configurazione del client, al momento accetta soltanto la proprietà `baseUrl` che indica l'URL di base della API `deception-core`.

È possibile specificare un file di configurazione personalizzato tramite il flag `-c`.

```
deception-cli -c <path-to-config> generate ...
```

5 Dockerfile

Uno dei compiti del componente `deception-core` consiste nel generare Dockerfile che nell'ultima versione venivano generati tramite semplice ma confusionaria concatenazione di stringhe.

Dato che questa operazione sarà sempre più frequente con l'aggiunta di nuovi componenti si è deciso di creare un modulo che permette di generare Dockerfile in modo object oriented, rendendo il processo più semplice e meno error-prone.

La classe `DockerfileBuilder` permette di aggiungere comandi che possono essere specificati in diversi modi:

- tramite la classe `CommandBuilder` che crea istanze delle singole classi comando
- come semplici stringhe per gestire eventuali casi limite

Le classi di supporto `CommandOptions` e derivate e `CommandOptionsBuilder` gestiscono le opzioni relativi ai vari comandi.

Lo strumento è progettato in modo da evitare errori nell'associazione delle corrette opzioni ai loro rispettivi comandi.

Si riporta di seguito un esempio di utilizzo:

```
builder.addCommand(CommandBuilder.from(baseImage, "22.0.5").name("builder"));
builder.addCommand(CommandBuilder.env("KC_HEALTH_ENABLED", enableHealth.toString()));
builder.addCommand(CommandBuilder.env("KC_METRICS_ENABLED", enableMetrics.toString()));
builder.addCommand(CommandBuilder.env("KC_HTTP_ENABLED", enableHttp.toString()));
builder.addCommand(CommandBuilder.workdir("/opt/keycloak"));
builder.addCommand(CommandBuilder.from(baseImage, "latest"));
builder.addCommand(
    CommandBuilder.copy("opt/keycloak/", "/opt/keycloak/"),
    CommandOptionsBuilder.copy().from("builder")
);
builder.addCommand(CommandBuilder.copy(configFile, "/opt/keycloak/export.json"));
```

In futuro verrà creata una suite apposita di script per automatizzare il processo di aggiornamento della libreria a seguito di modifiche alla specifica Dockerfile.