Progetto Programmazione ad Oggetti
"DockerComposer"

Tommaso Sotte Matricola **1122283** Università di Padova

Anno 2016/2017

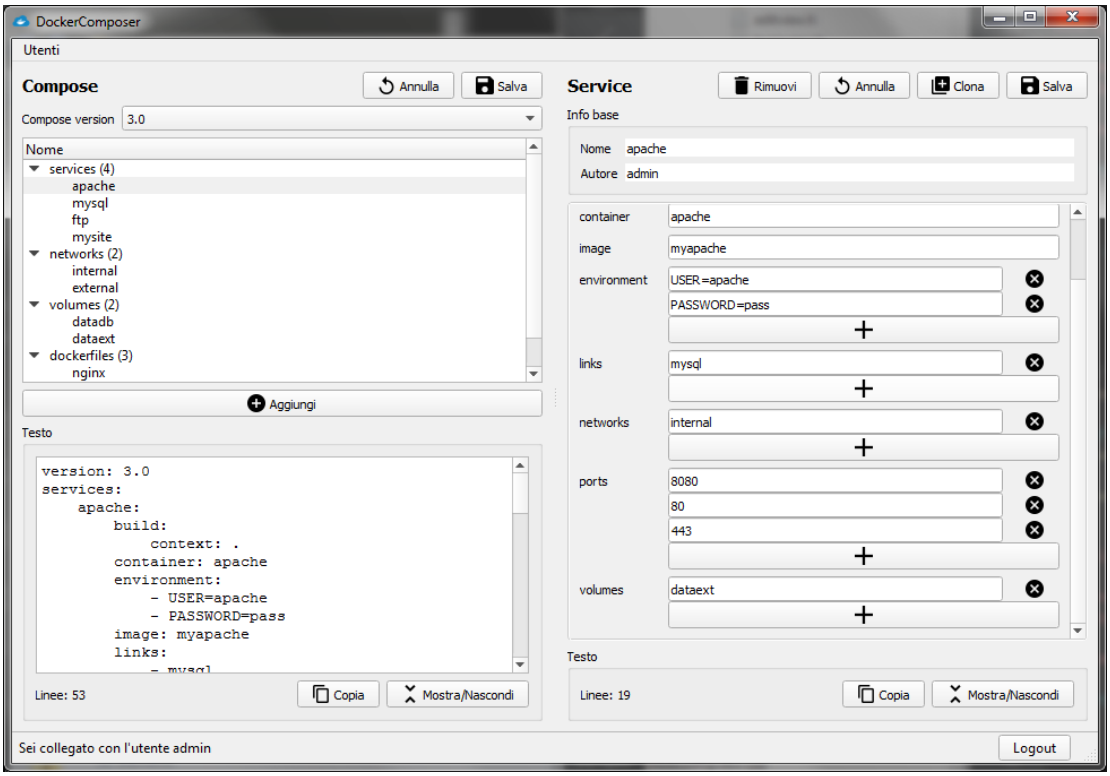

Figura 1: Anteprima del programma

1. Scopo del progetto

Il progetto DockerComposer vuole fornire una interfaccia utente per la creazione, lettura e modifica di file `docker-compose.yml` (o "compose"), un file di configurazione usato da Compose, un tool di Docker.

Docker è un progetto open-source che automatizza il processo di deployment di applicazioni (o "service") all'interno di container software, gestendo autonomamente le risorse in un ambiente incapsulato. Compose (Docker Compose) è un tool che si occupa di definire l'interazione fra i componenti e le risorse di docker (o "parti del compose") in un'unica configurazione, semplificandone l'utilizzo. È dunque essenziale per uno sviluppo più veloce in ambiente di testing automatizzati o di sviluppo di software (per isolarlo e limitare le risorse).

È possibile connettere un service con altre parti dello stesso compose. Ogni parte appartiene ad una delle seguenti categorie: Service, Network o Volume (descritte più in dettaglio successivamente).
Solitamente i file compose vengono modificati manualmente via un editor di testo. Le funzioni aggiunte dall'applicazione rispetto ad un editor di testo sono: i suggerimenti dell'autocompletamento per il riferimento di altri parti all'interno del compose, la generazione di un file YAML valido, feedback visivi di alcuni problemi comuni, la gestione di utenti con diversi permessi per collaborare sullo stesso compose, l'esportazione di singole parti e l'unicità dei nomi delle parti. La finalità dell'applicazione è quella di produrre un output di testo della configurazione compose valida.

La gestione degli utenti è delegata dall'utente "admin" (sempre presente di default), il quale è l'unico a poter creare/rimuovere altri utenti ed assegnare loro un ruolo, e i relativi permessi.
Gli utenti sono suddivisi in base ai permessi, in 5 classi, Admin, Manager, Complete, Network e Volume, in base a quali parti possono modificare e se solamente le proprie o anche create da altri utenti.

2. Descrizione delle gerarchie di tipi usate

La suddivisione delle gerarchie dei tipi di classi presenta due blocchi: model e view.
Nei model, viene fornita una classe base astratta, da fondamenta sia per user che per compose, `DataModel`, che definisce il "modello astratto". È stato evitato appositamente l'utilizzo di Qt nelle classi model per fornire la massima compatibilità con le STL e per poter utilizzare un framework diverso da Qt, nel caso si volesse cambiare.
È stato scelto il formato JSON per i dati perchè ritenuto più semplice ed immediato del formato XML suggerito da Qt. Inoltre è stata usata una libreria esterna[1], rispetto al supporto integrato di JSON in Qt.

[1]Libreria JSON: github.com/nlohmann/json, con licenza MIT, è una libreria caratterizzata da una sintassi immediata, supporto alla STL e una semplice integrazione. Presenta un leggero overhead, non ponendosi le prestazioni come obiettivo primario.

2.1 Compose/Docker

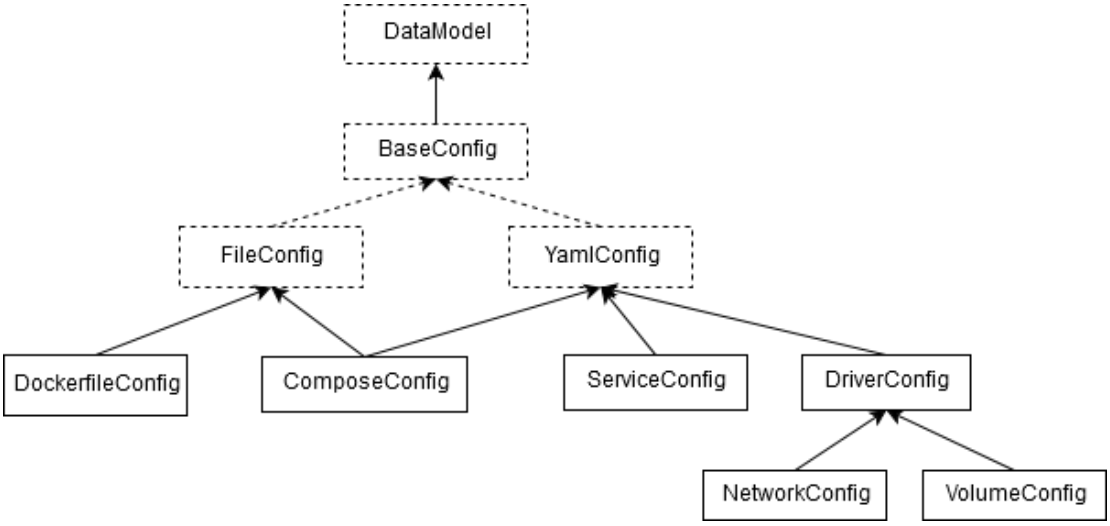

Figura 2: Grafico della gerarchia di parti del compose

La struttura del `docker-compose.yml` è stata implementata seguendo il "Compose file reference"[2] ufficiale, nella versione attuale, 3. Si noti che non sono stati implementati tutti i parametri disponibili[3], ma abbastanza per garantire le funzionalità principali e permettere la creazione di un compose funzionante.

La classe astratta derivata da `DataModel` è `BaseConfig` e rappresenta il modello di una configurazione generica, intesa solitamente come singolo file oppure come parte di un altra configurazione. È identificata da un *name* ed un *author*. Offre un metodo per validare, ad esempio la sintassi o i valori dei parametri, usando le eccezioni, ed un metodo per ottenere la configurazione sottoforma di testo.

Dalla config. base segue la suddivisione in `FileConfig`, che specifica configurazioni relative ad un singolo file, con il campo *filename*, e in `YamlConfig`, che identifica una configurazione YAML, che può essere una parte di una configurazione completa, e dispone del metodo `YamlFile getYaml()`[4] per favorire la l'innestamento delle parti del compose. Quest'ultime derivano virtualmente da `BaseConfig` per permettere l'ereditarietà multipla.

La classe `ComposeConfig` è una configurazione YAML su un singolo file (deriva sia da `FileConfig` che da `YamlConfig`). Ha una versione e conterrà il contenitore principale del progetto (di tipo `DockerParts`) composto da Service, Network e Volume.

`DockerfileConfig` invece è un singolo file contenente del *text*, di supporto al progetto e non essenziale nell'output finale; è utile per avere un riferimento del nome del Dockerfile in un service e per una validazione basica della sintassi, realizzata con regex.

[2]Docker compose file reference: https://docs.docker.com/compose/compose-file/

[3]Nota sui parametri non implementati: La maggior parte dei parametri non considerati avrebbero avuto la stessa identica implementazione di quelli presenti, ad eccezione di singolari casi più complessi.

[4]YamlFile: è classe creata appositamente per generare del testo compatibile con le specifiche *YAML 1.2* (http://www.yaml.org/spec/1.2/spec.html). Sono stati implementati degli `std::pair` appositi e un overloading, con vari override per ogni pair, dell'operatore `<<`. Supporta i pair con key `std::string` e valore del tipo `std::string`, `std::vector<string>` o `std::map<string, string>`.

Le parti del compose, Service, Network e Volume, sono caratterizzate da una derivazione da `YamlConfig`, per permette l'innestamento dell'output in YAML del compose, su `ComposeConfig`.

La classe`ServiceConfig` descrive un singolo servizio, di cui è richiesto specificare un *image* (ovvero immagine di un container) o il percorso per una *build*. Opzionalmente è possibile specificare i collegamenti con le altre parti del compose, un *dockerfile* (per nome) ed altri parametri. `DriverConfig` è una classe che serve solamente a raggruppare parametri simili, come il driver o il fatto che sia esterno, che si possono ritrovare sia in `NetworkConfig` e `DriverConfig`, dei quali esso è classe base. `NetworkConfig` specifica una rete, solitamente intesa come un gruppo, del quale i Service possono farne parte.
Infine `VolumeConfig` rappresenta un volume, inteso come dati, solitamente collegato ad uno o più servizi, usato da Docker per l'accesso multiplo alle stesse risorse.

2.2 Utenti

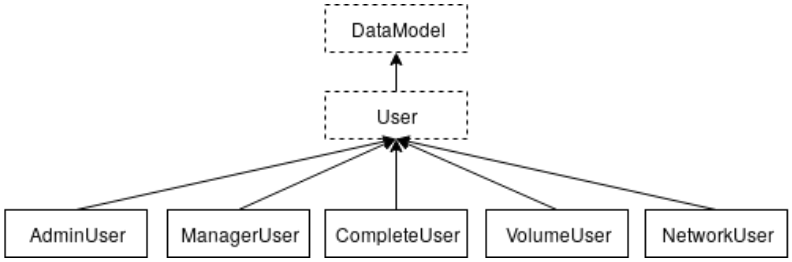

Figura 3: Grafico della gerarchia degli utenti

La gerarchia di utenti ha come base la classe astratta `User`, con metodo di clonazione virtuale puro, la quale a sua volta deriva dalla classe `DataModel`.

Ogni utente è identificato da un *name*, univoco nel contesto della lista di utenti, ha una *password*, per effetturare l'accesso, e un set di permessi, realizzato attraverso delle specifiche funzioni virtuali (es. `bool canEditService()`). È stato scelto di utilizzare delle funzioni al posto di campi booleani perchè i permessi sono relativi alla classe stessa e sarebbero stati al più valori costanti, e quindi si sarebbe copiati per ogni utente di tale classe, portando ad uno spreco di memoria.

Le label dei ruoli degli utenti sono impressi nella variabile statica `const std::string modelName`, unica per ogni classe, ed ottenibili via polimorfismo tramite la funzione `getRole()`, la quale fa accesso a `getModelName()`, implementazione presente in ogni classe concreta derivata da `DataModel`. Si è scelto di usare un campo statico perchè questo permette una chiaro riscontro del ruolo e la possibilità di cambiare nome del ruolo senza dover modificare stringhe nel resto del progetto, dove si effettuano le comparazioni.

I permessi sono: editare parti Service (di default *true* su tutti i ruoli); editare Network, specifico per i NetworkUser, altrimenti sempre possibile; editare Volume, come per i Network, è specifico per i VolumeUser; editare tutti i componenti, abilitato per i Complete, Manager o Admin user; editare anche le parti degli altri, solamente possibile ai Manager o Admin user; essere Admin, via `isAdmin()`, il quale permette completo accesso all'applicazione, ed ha tutti i permessi di un Manager.

2.3 Contenitori

I contenitori presentano una semplice gerarchia. Come base abbiamo `DataList`, la quale deriva anch'essa da `DataModel`.

`DataList` è una classe contenitore templetizzata, di cui il primo parametro `class T` indica il tipo di elementi contenuti. I dati internamente sono organizzati e gestiti da una `std::list`. Si è scelto di usare la lista invece del vettore perchè permette l'inserimento e la rimozione casuale più veloce.

È organizzata per fare la copia profonda e la distruzione profonda degli elementi che contiene. Oltre a reimplementare la funzione di export, mette a dispozione i metodi base per l'aggiunta, rimozione e visione degli elementi, ed espone gli iteratori della lista contenuta, per una facile consultazione.

Gli utenti sono contenuti nella classe derivata `UserList`, la quale contiene elementi `User*`, specificati nella derivazione da `DataList<User*>`. È stato reimplementato il metodo di import ed un metodo statico di factory per la creazione degli utenti. Presenta ulteriormente una funzione di ricerca degli utenti via nome (univoco all'interno del contenitore).

Ugualmente, per le parti del compose, troviamo, sempre derivata da `DataList`, la classe `DockerParts`, composta da elementi `BaseConfig*`. Anche essa reimplementa il metodo di import e un metodo statico di factory per la creazione di parti. Di fondamentale utilizzo, sia per la GUI che per l'export in formato YAML, c'è il metodo per filtrare le parti in base al tipo, la quale restituisce una copia profonda della lista con solamente le parti filtrate.

La classe `DockerParts` è contenuta a sua volta all'interno dalla classe `ComposeConfig`, perchè essenzialmente le parti (gli elementi) costituiscono il compose. Per lo scopo del progetto avremmo quindi un solo `ComposeConfig`, ma questa decisione permette di poter estendere in futuro il supporto ad ulteriori compose.

2.4 Views

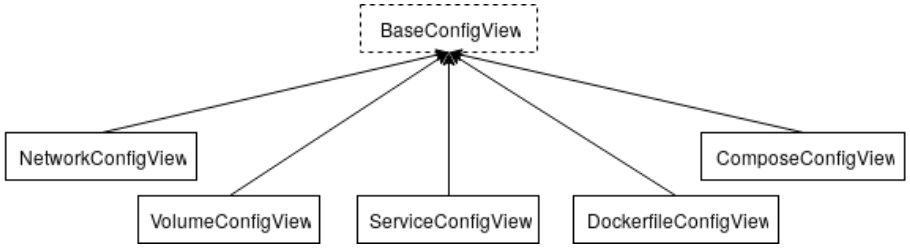

Figura 4: Grafico delle views del compose

Le classi views seguono la stessa composizione delle classi modello compose. Tuttavia sono state semplificate e rimosse le classi non necessarie.
La classe base astratta `BaseConfigView`, è resa astratta dal metodo virtuale puro `setupEditWidget()`. Questa classe offre l'essenziale per creare una view per BaseConfig, e permette di essere derivata e composta (più in dettaglio al capitolo 3.3)

3 Codice Polimorfo

La classe base astratta fondamentale `DataModel`, essendo base di ogni modello, ha il distruttore virtuale cosicchè la distruzione di una sua derivata distrugga correttamente l'oggetto ed eviti dei memory leak qualora venga usato un puntatore a `DataModel` (nel progetto verranno utilizzati più concretamente i puntatori ad `User` e `BaseConfig`).

`DataModel` raggruppa dei metodi comuni che sono stati resi per l'appunto virtuali a contratto puro, ovvero `DataModel* clone()`, che crea un nuovo puntatore copiando l'oggetto stesso, ed i metodi di import/export dal formato dei dati JSON, rispettivamente `void importFromJSON(nlohmann::json&)` e `nlohmann::json exportToJSON() const`.

3.1 Compose

Le classi che fanno parte della gerarchia dei compose, sempre derivando da `DataModel`, ottengono innanzitutto il metodo polimorfico `BaseConfig* clone()`.

Importante per un qualsiasi file di configurazione abbiamo il metodo virtuale `bool validate()`, implementato in modo specifico per ogni tipo derivato, con lo scopo di effettuare dei controlli sulla configurazione, idealmente prima del suo utilizzo.
Ancora, sempre importante, abbiamo il metodo `std::string getText()`, che ritorna l'output della configurazione in formato di testo.
Sulle parti compose, quest'ultimo metodo viene modificato dalla classe `YamlConfig`, e dunque le sue classi derivate otterrano il testo passando per il metodo virtuale `YamlFile getYaml()`; una classe `YamlFile` mette poi a disposizione un metodo `std::string getText()`, appunto utilizzato dal metodo `getText()` modificato da `YamlConfig`.

3.2 Utenti

Nella gerarchia delle classi utente si fa uso del polimorfismo innanzitutto sul metodo `User* clone()`. Invece i permessi sono dati dai metodi virtuali `bool canEditService()`, `bool canEditNetwork()`, `bool canEditVolume()`, `bool canEditOtherUsers()`, `bool isAdmin()`, i quali ritornano la possibilità o meno di avere quel permesso, e serviranno all'interfaccia per abilitare o disabilitare una determinata funzionalità. Di ciò se ne occupa il metodo virtuale `void setupPermissions()`, presente su ogni classe derivata da `BaseConfigView`, per ottenere una granularità dei permessi e la possibilità di estenderli.

3.3 Views

Come specificato in precedenza, la classe base astratta delle view per le parti del compose è `BaseConfigView`. In essa, in fase di progettazione della GUI, è stata fatta la scelta di renderla componibile liberamente, quindi con un costruttore vuoto e rendendo la classe astratta.

La composizione è realizzata attraverso dei metodi virtuali di "setup". Ognuno di esssi si occupa di costruire una parte dell'interfaccia. In dettaglio: `void setupHeaderWidget()` configura i pulsanti per annullare (le modifiche), clonare, salvare o rimuovere la configurazione; `void setupInfoWidget()` per mostrare il nome e l'autore; `void setupOutputWidget()` per l'output di testo; `void setupEditWidget()`, a contratto puro, il quale è implementato diversamente per ogni parte del compose; infine `void setupMainLayout()` per il layout del widget stesso e

`void setupPermissions()` per i permessi, abilitando e disabilitando parti dell'interfaccia in base ai permessi dell'utenti loggato.

Sempre sulla stessa classe troviamo i metodi virtuali: `BaseConfig* getConfig()` per ottenere la config mostrata; `void save()` chiamata al salvataggio dei dati dalla GUI al modello; `void reload()` per ricaricare i dati dal modello alla GUI; `void reloadOutput()` mostra il testo generato dalla config via `getText()` (su `BaseConfig`); `bool validate()` solitamente chiamata dopo il salvataggio, mostra eventuali errori sulla GUI; `void resetErrors()` pulisce gli errori mostrati; `void showErrors(const QString& msg)` mostra gli errori, solitamente successivi ad una validazione.

4. Manuale Utente

I dati sono salvati divisi, seguendo la struttura dei modelli, in utenti, nel file `users.json`, ed in parti del compose, nel file `docker-compose.yml.json`, e si trovano nella root del programma. Se non presenti verranno generati automaticamente vuoti, e nella lista utenti si potrà trovare l'utente *admin* con la password di default "admin".
Nota: Ogni funzione di salvataggio ed aggiunta di parti, che sia con la clonazione o con l'aggiunta di una nuova parte, salverà su disco.

Una volta effettuato l'accesso si arriva alla finestra principale, divisa in *ComposeView* (che mostra il compose) a sinistra, e *RightView* (o modifica delle parti del compose) a destra.
Nella pannelo *ComposeView*, che mostra il compose attuale, è possibile aggiungere nuove parti tramite il pulsante "Aggiungi", salvare e annullare le modifiche o visualizzare l'output completo. Nella lista è possibile scegliere quale parte modificare. Una volta scelta una parte con un singolo click è possibile modificarla nella *RightView*.
Sulla *RightView* è possibile editare una parte scelta dalla *ComposeView*. In alto, sull'header, abbiamo la possibilità di rimuovere, annullare le modifiche, clonare o salvare. Ogni parte è caratterizzata dall'intenstazione dove possiamo vedere il nome della parte e l'autore.
Nel corpo della view troviamo i vari parametri modificabili di quella parte del compose. Una volta effettuate le modifiche, salviamo, ed eventuali errori verranno mostrati in alto. Sempre al salvataggio, l'output, che possiamo trovare sotto, verrà aggiornato con le modifiche.

Sul menu in alto troviamo l'azione *Utenti* che apre un dialogo per visualizzare la lista di utenti. L'utente corrente è evidenziato in color cyano. Un utente può solamente modificare la proprio password.
Solo un utente admin ha la possibilità di modificare la password di tutti gli utenti, crearne di nuovi oppure rimuoverli. Non è possibile cancellare l'utente con cui si è loggati.

Permessi: quale tipo di parte del compose è possibile modifica in base al ruolo dell'utente. *Nota: solo gli utenti admin hanno la possibilità di gestire altri utenti.*

Ruolo	Dockerfile	Service	Network	Volume	—	Di altri utenti*	Clonazione**
Admin	si	si	si	si		si	si
Manager	si	si	si	si		si	si
Complete	si	si	si	si		no	si
Network	si	si	si	no		no	no
Volume	si	si	no	si		no	no

*: indica la possibilità di modificare parti del compose non create dall'utente loggato.
**: la clonazione per Volume e Network è limita a, rispettivamente, Volume e Network.

5. Ore utilizzate

Progettazione Modello	Progettazione GUI	Codifica Modello	Codifica GUI	Debugging	Testing
4h	6h	8h	35h	6h	2h

Ore aggiuntive: 2.30h per `EditableStringList`, non avendo trovato un widget Qt simile; 2h per `YamlFile`, necessario per l'output in YAML necessario al programma.

6. Ambiente di sviluppo

- **Sistema Operativo:** Microsoft Windows 7 SP1 64-bit
- **Compilatore:** gcc 5.3.0
- **Qt:** 5.9.1
- **Qmake:** 3.0