UNIVERSITÀ DEGLI STUDI DI MESSINA

DIPARTIMENTO DI SCIENZE MATEMATICHE E INFORMATICHE,
SCIENZE FISICHE E SCIENZE DELLA TERRA

Corso di Laurea Triennale in Informatica

Docktor: implementazione di un servizio web che sia un monitor per Docker containers

Docente:
Prof. Andrea Nucita

Studente:
Claudio Anchesi, 513605

ANNO ACCADEMICO 2022/2023

Indice

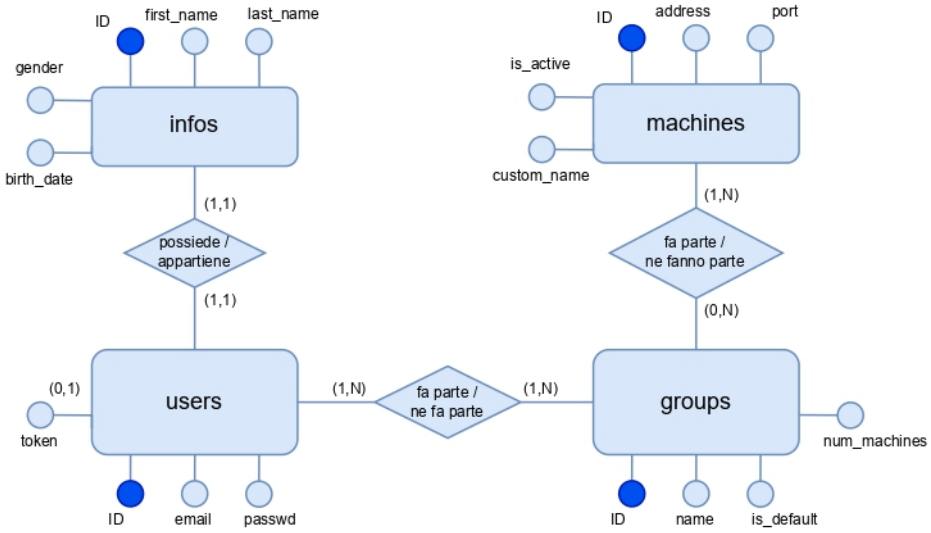
1 Introduzione

Come suggerisce il titolo del presente scritto, il progetto esposto vuole essere un servizio web che permetta una non complessa gestione di Docker containers, con la possibilità di crearne, eliminarne e gestirne l'esecuzione, presenti su più macchine della rete, queste concentrabili in gruppi. A tal fine, il progetto è diviso in due componenti (tre se si include il database), entrambe pubblicate su GitHub[1] e su Docker Hub[2]: un *agent*, che permette la comunicazione tra il servizio web e le macchine in questione, e l'*app*, il servizio in sé, che mette a disposizione un'interfaccia web per l'utente. Entrambi possono essere istanziati a loro volta come containers, il primo sulle macchine delle quali si vuole avere l'accesso ai suddetti, e il secondo sulla *workstation* che metterà a disposizione il web server.
Per la realizzazione del back-end dell'*app* e dell'*agent* si è utilizzato *NodeJS*, un software per l'esecuzione di codice Javascript esternamente ai browser, e quindi tale da rendere quest'ultimo un linguaggio di programmazione a sé stante, basato sul motore V8 di Google Chrome. La scelta di questo, rispetto ad un più comune PHP, è dovuta alla voglia del candidato di sperimentare un nuovo linguaggio di programmazione (Javascript) in tutte le sue sfaccettature, front-end e back-end. Si è inoltre adoperato un database relazionale di tipo *MySQL* per la persistenza dei dati necessari al funzionamento, anch'esso istanziato come container e utilizzabile tramite file docker-compose rintracciabili su link di GitHub di cui sopra; la comunicazione con il back-end è stata effettuata tramite modulo open source di NodeJS *sequelize*, il quale adotta un concetto architetturale di tipo MVC, model-view-controller. Per la realizzazione del servizio web sono state create ed utilizzate delle *RESTful API*, gestite dal modulo open source di NodeJS *express*, che permette una suddivisione per URI, metodo della richiesta HTTP, e l'utilizzo di middleware per una più efficiente gestione delle routes.
Per concludere, il front-end è stato realizzato tramite la scrittura di pagine HTML, AdminLTE per lo stile, un framework di CSS derivato da Bootstrap, e quasi esclusivamente *jQuery* per la gestione degli script Javascript.

[1] Link alla repository di Github
[2] Link alle repositories di Docker Hub: docktor, docktor-agent e docktor-db

2 Database

Il DBMS utilizzato per questo progetto è MySQL, scelto per via della semplicità di utilizzo, non estesa quantità di dati da gestire (non si parla di Big Data) e per una più rigida gestione delle relazioni tra i dati. In realtà, è presente un'immagine apposita presente a piè di pagina della sezione 1, ma presenta in più dell'immagine ufficiale di MySQL il caricamento del database. Di seguito lo schema ER:



Le relazioni *N a N* presenti si sono risolte logicamente con l'utilizzo di due tabelle aggiuntive che le gestissero mettendo in relazione gli id dell'utente e del gruppo, per la relazione tra le entità *users* e *groups*, e gli id del gruppo e della macchina, per la relazione tra le entità *groups* e *machines*. Per una più semplice gestione si è pensato di non creare utenti aggiuntivi per il database ma utilizzare i permessi di root, nonostante sia evidente la mancanza di sicurezza di tale scelta.

Lo schema concettuale riportato fornisce un'idea sulla gestione delle informazioni: ad un utente corrispondono una serie di informazioni personali, non utili ai fini del progetto ma interessanti per fini di profilazione; alla sua registrazione viene automaticamente creato un gruppo di *defaalt* (identificato dal campo *is_ default*) al quale verranno aggiunte le macchine che l'utente vuole gestire, ed è inoltre possibile la creazione di gruppi aggiuntivi, permettendo una suddivisione delle suddette per una maggiore facilità di controllo.

Ad un utente è associato un *token* per l'autenticazione dopo il login, creato nel back-end tramite il modulo open source di NodeJS *jwt*, acronimo che sta per *JSON Web Token*, tale da permettere la creazione di un token a partire dalle informazioni dell'utente, più un sale. Le macchine sono caratterizzate da un nome, un indirizzo (IPv4, IPv6 o dominio), una porta e un flag che ne rappresenta l'attività.

Nel back-end, il modulo *sequelize*, il quale opera secondo il modello *model-view-controller*, ha permesso la creazione di un modello per tabella che la rappresentasse, tramite specifica di tutti i campi e le loro caratteristiche, e di un controller per

```
const DataTypes = require('sequelize').DataTypes;
const sequelize = require('../utils/dbConnect');

// Modello per la tabella users
const User = sequelize.define('user', {
    id: {
        type: DataTypes.INTEGER,
        primaryKey: true,
        autoIncrement: true
    },
    email: {
        type: DataTypes.STRING,
        allowNull: false,
        unique: true
    },
    passwd: {
        type: DataTypes.STRING,
        allowNull: false
    },
    token: {
        type: DataTypes.STRING(1024),
        allowNull: true
    }
}, {
    timestamps: false,
    createdAt: false,
    updatedAt: false
    });
```

```
const updateUser = async (req, res) => {
    const transaction = await sequelize.transaction();

    try {

        // Se è stato passato un nuovo password, la cripta
        if (req.body.passwd)
            var passwd = bcrypt.hashSync(req.body.passwd, await bcrypt.genSalt(10));

        // Aggiorna i campi passati
        await User.update({
                email: req.body.email || req.user.email,
                passwd: passwd || req.user.passwd
            }, {
                where: {
                    id: req.params.id || req.user.id
                }
            }, { transaction });
        await transaction.commit();
        res.status(200).send("Modifica avvenuta con successo");
    } catch (error) {
        await transaction.rollback();
        sendError(error, res);
        return;
    }
}
```

Figura 1: Modello e Controller di Sequelize

modello nel quale definire le operazioni sulle tabelle, successivamente usate per le API. Di seguito un esempio di modello e funzione del controller dello stesso:

3 Back-end

3.1 Introduzione alle RESTful API

Per la comunicazione col database e le funzionalità di popolazione e calcolo del front-end sono state create delle API basate sul paradigma REST, il quale consiste in una serie di principi architetturali fondati sull'identificazione di una risorsa tramite un URI, alla quale è possibile associare delle operazioni, definite nuovamente dall'URI in questione e dal metodo HTTP utilizzato per il raggiungimento di tale risosrsa. Questi ultimi, inoltre seguono una mappatura con le operazioni *CRUD*, come segue nella tabella:

HTTP	CRUD
`POST`	`Create`
`GET`	`Read`
`PUT`	`Update`
`DELETE`	`Delete`

Un esempio, preso dal codice del progetto in questione, è l'utilizzo di una richiesta `POST` ad un URI del tipo `https://localhost:8443/users/create`: così facendo si sta comunicando al server che si vuole operare sulla risorsa `users`, che corrisponde alla tabella degli utenti nel database, e che si vuole effettuare una creazione, un *INSERT*, avvalorato dalla richiesta `POST` e dall'ulteriore parte `create` presente nell'URI.

3.2 Implementazione delle API

Come accennato nella sezione 1, per l'implementazione dell'API è utilizzato il modulo *express* di NodeJS, il quale permette automaticamente una gestione degli URI e dei metodi. Di seguito un esempio di gestione di express sulla prima parte dell'URI:

```
// ------- Routes -------
api.post('/register', alreadyExists('user', false), createUser);

api.use('/', verifyToken)
api.use('/users', require('./users'));
api.use('/groups', require('./groups'));
api.use('/infos', require('./infos'));
api.use('/machines', require('./machines'));
```

Figura 2: API con express

L'oggetto `app` corrisponde ad un'istanza della classe `Router` di express, il quale presenta i metodi corrispondenti alle richieste HTTP (come il metodo `post()` presente come primo) e il metodo `use()`, molto utile per operazioni indipendenti dal metodo della richiesta. I primi argomenti dei sopracitati metodi rappresentano la parte dell'URI a cui fanno riferimento; gli argomenti successivi sono *callback functions*: queste richiedono due argomenti, uno rappresentante la richiesta HTTP, con header, query, body e altre componenti, e l'altro rappresentante la risposta, con header, body e metodi per la costruzione. Come si evince dalla prima chiamata, è possibile passare più di una *callback function*: tutte queste, esclusa l'ultima in ordine di passaggio, presentano un terzo parametro, `next`, che permette il passaggio alla funzione successiva; questi si chiamano *middleware*. Nella prima chiamata `post()` si può vedere un middleware `alreadyExists()`, definito per verificare l'esistenza di

un certo elemento nella tabella del database precisata come argomento. Le funzioni passate tramite i `require()` sono altri oggetti `Router` definiti in altri file per gestire molteplici richieste su un determinato URI.
Un middleware importante utilizzato è `verifyToken()`, come riportato nella prima chiamata `use()` nell'immagine di cui sopra: esso compie il lavoro di verifica del token fornito dopo il login, ed è così utilizzato su tutti gli URI a seguire (per via del metodo `use()`), così da permettere l'utilizzo delle API solo a login effettuato.
Inoltre, dall'immagine 1 del modello `user` presente nella sezione 2 si può notare come la funzione `updateUser()` sia, per l'appunto, una *callback function* di quelle appena spiegate, precisamente utilizzata per la chiamata `PUT` all'URI `/user/this`, grazie al quale si possono aggiornare i dati di accesso relativi all'utente corrente.

Tutte le API seguono la filosofia di cui sopra. Ve ne sono alcune non riservate all'utente che ha effettuato l'accesso, e quindi senza middleware `verifyToken()`, che aiutano nel caricamento di componenti HTML, CSS e Javascript nel frontend, come evincibile dall'immagine seguente:

```
app.use('/templates', templates);
app.use('/scripts', scripts);
app.get('/style', (req, res) => {
    res.sendFile(join(__dirname, '../public/style/style.css'));
})
```

Figura 3: Altre API

Questi si occupano rispettivamente, di restituire al client delle parti di codice HTML riutilizzato, il foglio di stile ed eventuali script in JS del frontend; questi ultimi due sono stati creati per evitare di inserire nei tag `<script>` e `<link>` i path reali dei file.
Ulteriori chiarimenti su API utilizzate sono rintracciabili nel codice della repository presente a piè di pagina nella sezione 1.

3.3 HTTPS

Si è scelto di implementare una comunicazione tramite protocollo HTTPS, utilizzando dei certificati *SSL*. Nel caso delle prove effettuate in fase di sviluppo si sono scelti dei certificati *self-signed*, il che rende possibile la comunicazione soltanto previa autorizzazione del browser visitando un URL dell'interfaccia web.

A tal fine si è utilizzato il modulo *https* di NodeJS, che rende il software un web server con protocollo HTTPS, tramite la lettura del certificato e della chiave SSL necessari. Di seguito il codice relativo:

La variabile `config` contiene i alcuni dati letti dalle variabili d'ambiente del web server. In questo caso vengono letti, tramite il metodo `readFileSync()` del modulo *fs*, i path dei file necessari di cui sopra. Infine, si avvia il server mettendolo in ascolto su una porta specificata, che nel caso dell'ambiente di sviluppo è la `8443`.

```
var server = https.createServer({
    key: fs.readFileSync("/app/ssl/key.pem"),
    cert: fs.readFileSync("/app/ssl/cert.pem"),
    passphrase: fs.readFileSync("/app/ssl/passphrase.txt").toString() || ""
}, app);
server.listen(config.port);
```

Figura 4: Connessione HTTPS

4 Front-end

Il front-end del servizio si compone, come anticipato nella sezione 1, di pagine scritte in HTML, con stile sia basato sul framework AdminLTE, derivato da Bootstrap, che su un file locale `style.css`; per gli gli script Javascript si è adottato l'utilizzo di jQuery, una libreria atta alla stesura di codice meno verboso.

Docktor si basa su soli sei ipertesti, generalmente contenenti poche se non nessuna informazione, in quanto questi verranno *popolati* dagli script jQuery grazie alle richieste AJAX alle API della sezione 3.2. Così facendo, il web server si occuperà soltanto di fornire le informazioni provenienti o derivabili dal database, lasciando la costruzione della pagina interamente al client.
Nonostante la natura delle richieste AJAX, come suggerito dal nome completo dell'acronimo (*Asynchronous Javascript And XML*, vi è stata la necessità di effettuare alcune operazioni lato client solo al compimento di una delle suddette, il che rende quest'ultima difforme dalla sua naturale predisposizione; difatti, tramite il metodo `then()` degli oggetti `Promise()`, come quelli restituiti da un AJAX, è possibile attendere il completamento di una richiesta asincrona, rendendola, praticamente, sincrona. Di seguito un esempio di richiesta AJAX atta alla popolazione di una porzione di pagina:

```
// Ottenimento informazioni utente
$.ajax({
    url: '/api/infos/this',
    type: 'GET',
    success: (data) => {
        $('#id').val(data.uid)
        $('#nome').val(data.first_name)
        $('#cognome').val(data.last_name)
        $('#data').val(data.birth_date)
        $('#gender').find('option').each((i, el) => {
            if(el.value == data.gender)
                $(el).prop('selected', true)
        })
    }
})
```

Figura 5: Richiesta AJAX

Le prime pagine con cui si avrà a che fare sono quelle dedicate al login e alla creazione di un nuovo utente; da queste pagine trapela la caratteristica multiutente di questo servizio, senza tuttavia definizione di un amministratore né interazione tra gli utenti.

Una volta effettuato il login si avrà a che fare con la Homepage di Docktor, che comprenderà delle informazioni circa le macchine registrate, i container rintracciabili da queste, ed eventuali volumi.

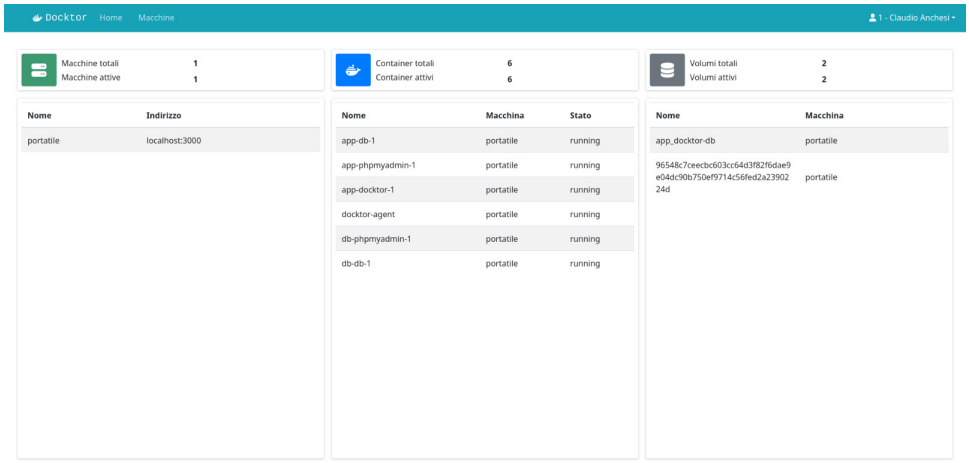


Figura 6: Homepage di Docktor

Da questa pagina ci si può spostare sulla sezione relativa al proprio profilo utente cliccando in alto a destra, dove presente il nome e il cognome. In quest'ultima si possono modificare le impostazioni dell'utente e creare eventuali gruppi per suddividere le macchine.

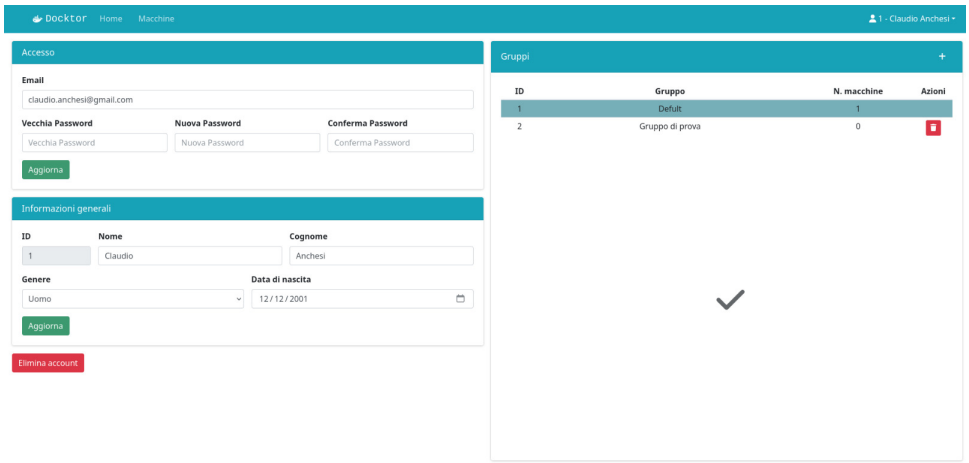


Figura 7: Profilo utente di Docktor

Dalla barra di navigazione in alto, si può tornare sulla Homepage o andare sulla pagina relativa alla gestione delle macchine. Da qui, si possono *registrare* nuove macchine (nelle quali deve essere istanziato il container dell'agent citato nella sezione 1), o aggiungerle ad eventuali gruppi secondari creati nella pagina precedentemente descritta.

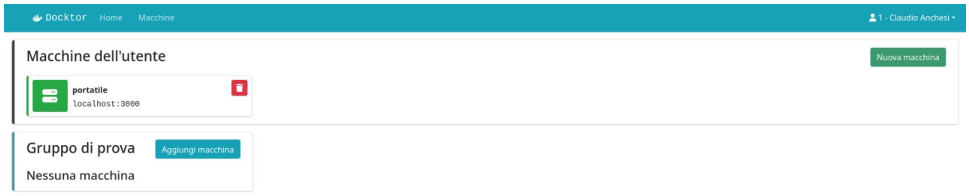


Figura 8: Pagina di gestione delle macchine

Le macchine colorate di verde vengono considerate Attive, ossia raggiungibili nella rete e con l'agent attivo, ciononostante, quando sarà necessario un collegamento con essa, ad esempio nella Homepage o nella sua pagina personale, al mancato raggiungimento verrà considerata Inattiva. è possibile anche eliminare le macchine dai gruppi secondari o totalmente.
Tramite il tasto vedre, o rosso, accanto al nome della macchina è possibile entrare nella sua pagina personale.

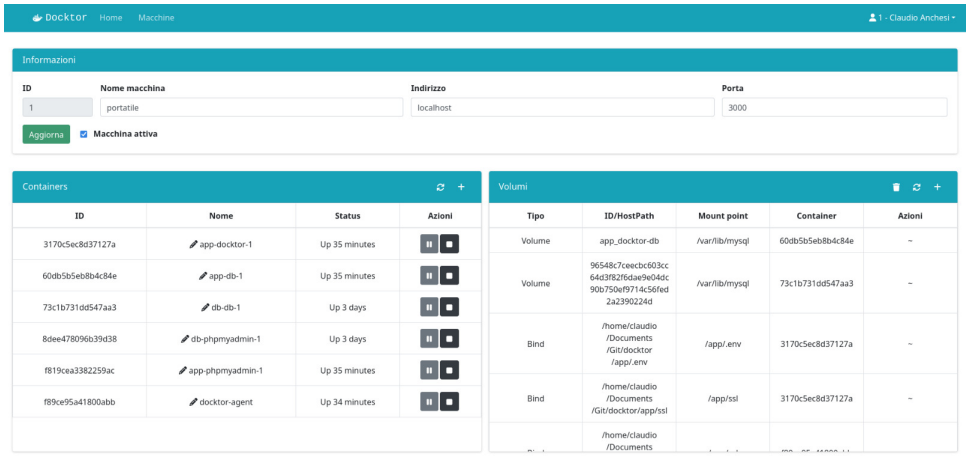


Figura 9: Pagina di gestione della singola macchina

Da questa schermata si possono modificare le informazioni per la connessione alla macchina, ed operare su containers e volumi presenti: per i containers è possibile metterli in pausa, fermarli, eliminarli, rinominarli, crearli con qualche restrizione o caratteristica, scaricare eventuali immagini non presenti; per i volumi è possibile crearli ed eliminarli, ma verranno visualizzati anche i binding dei containers. All'aggiornamento dei containers, possibile con l'icona apposita sopra la tabella, verranno aggiornati anche eventuali volumi automaticamente creati.

5 Docker

5.1 Agent

L'*agent* è un software semplice realizzato in NodeJS che funges da *socket* per Docktor, in quanto è ciò che permette una relazione tra il servizio web e la reale socket di Docker presente nelle macchine Linux che si vogliono controllare.

A sua volta, l'agent è istanziabile come container, tramite l'immagine presente nel piè di pagine della sezione 1. Per un corretto funzionamento è possibile eseguire un file `docker-compose.yaml`, presente nel repository GitHub citato nello stesso piè di pagina. Oltre a ciò, è necessaria una directory `ssl/` con all'interno i tre file per la connessione HTTPS, ossia `key.pem`, `cert.pem` e `passphrase.txt`, che sarà poi passata al container tramite binding.

Il software effettua solamente una operazione: esso riceve, seguendo paradigma REST, delle richieste HTTP alla medesima maniera spiegata nella documentazione delle API di Docker[3], modificando eventuali headers e inoltrandola alla Unix socket del demone di Docker, anch'essa collegata al container dell'agent tramite binding; la risposta è poi inoltrata al client di Docktor.

N.B. Potrebbe essere necessario raggiungere la macchina desiderata prima tramite il suo indirizzo in una scheda del browser specificando il protocollo HTTPS, così da fornire al browser la fiducia verso i vertificati SSL *self-signed*.

5.2 App

Il servizio web di Docktor è anch'esso istanziabile come container tramite l'immagine rintracciabile nel link nel piè di pagina nella sezione 1.
Come per l'agent, vi è un file `docker-compose.yaml`, presente nella repository del link di GitHub, che permette una più semplice istanziazione, e sono necessari due elementi aggiuntivi da provvedere autonomamente:

- una directory `ssl/` con all'interno i tre file per la connessione HTTPS, ossia `key.pem`, `cert.pem` e `passphrase.txt`, che sarà poi passata al container tramite binding;
- un file `.env` che presenti le seguenti variabili d'ambiente:
  - `DB_HOST=<nome-servizio-db>`
  - `DB_PASS=<password-di-root>`
  - `DB_PORT=<porta-esposta>`
  - `PORT=<porta-del-servizio>`
  - `TOKEN_KEY=<token-per-criptazione-password>`

Le prime quattro variabili vanno anche modificate nel `docker-compose.yaml`.

Una volta che tutto è pronto basta eseguire il file come previsto da Docker e, dopo qualche secondo per via del caricamento del database, bastera recarsi nell'URL `https://<IP>:<PORT>`.

[3]Link alle API del demone di Docker