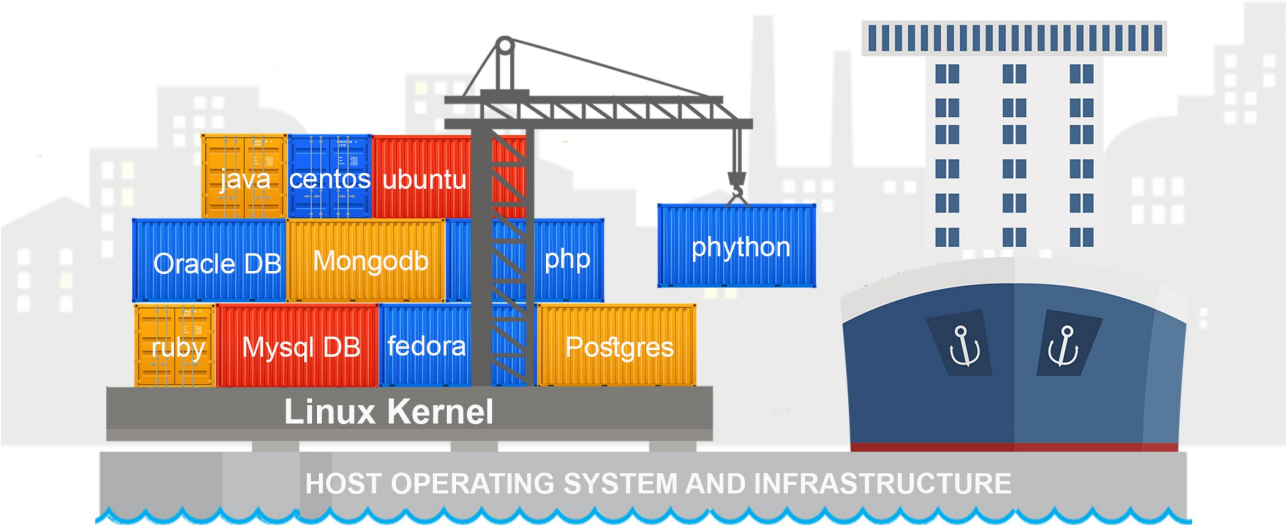
Docker è una piattaforma Open Source per lo sviluppo, il deploy e il running. Per applicazioni si intende genericamente applicazioni comunque complesse, come sono in genere i sistemi informatici aziendali. Applicazioni composte, ad esempio, da più servers e da più databases. **Simulare queste applicazioni su unica macchina implicherebbe un enorme dispiego di risorse.** Si supponga l'esistenza di un applicativo web composto da un server Tomcat che espone dei servizi e persiste i dati su un database MYSQL, il quale server a sua volta necessita di chiamare dei servizi residenti su altro server Jboss che persiste i dati su altro database Oracle. Nelle varie fasi di sviluppo software pertanto servirebbe installare e configurare sulla stessa macchina due servers e due database. Questo per i vari ambienti di sviluppo ed eventualmente test. Si supponga inoltre che si voglia fornire l'applicativo al cliente o ad altri colleghi del team di sviluppo per dei tests. Si dovrebbe provvedere ad installare e configurare due servers e due databases sulla macchina dei colleghi o clienti. Si immagini inoltre, come spesso accade nella realtà, che in produzione(ma anche in sviluppo e collaudo) il server 1 stia su una macchina dedicata, il server 2 stia su altra macchina, il database 1 stia su altra macchina e il database 2 su altra macchina ancora. Come può, in questo caso, uno sviluppatore fare dei test sulla propria macchina che simulino tutto l'applicativo? A questo scopo sono nate le macchine virtuali, che rappresentano appunto una macchina simulata, definita Guest. che viene lanciata su altra macchina, definita Host. Nella pratica si pensi di possedere un PC dotato di sistema operativo Windows 10(Host), sul quale si vuol lanciare il sistema operativo Linux(Guest), quindi la macchina virtuale Linux, oppure una VM che simuli una istanza di Windows 7(Guest) appunto su Windows 10(Host). WMWare, fra i tanti, fornisce appunto degli applicativi che sono delle macchine virtuali capaci di simulare una macchina Windows su Linux e viceversa. Un classico esempio di macchina virtuale è inoltre la JVM che praticamente simula una macchina reale, microcodice + chiamate di sistema, garantendo la portabilità del bytecode delle classi Java. Lanciare

un sistemo operativo Guest su un altro sistema operativo che rappresenta il sistema Host, implica l'uso di molte risorse hardware e software per il sistema Host. Un a macchina virtuale lanciata su un sistema Host, sul quale addirittura devono essere lanciate più macchine virtuali(Guests), appesantisce la macchina Host in modo non indifferente. Componente fondamentale di un gestore di macchina virtuale è l' **hypervisor** che è un software che gira sul sistema Host e si preoccupa di trasformare il microcodice del sistema Guest in microcodice del sistema Host per l'esecuzione. Un chiamata di sistema del Guest viene prima trasformata nel microcodice del Guest, poi trasformata nel microcodice dell'Host ed eventualmente in chiamata di sistema dell'Host. Il sistema Guest in termini di microcodice e di chiamate di sistema, è totalmente duplicato e interamente residente sull' Host. Questo è in estrema sintesi il concetto di macchina virtuale molto utile per la simulazione di sistemi complessi.

Container Management

In molti casi quindi sarebbe più opportuno e performante avere un componente capace di trasformare subito la chiamata di sistema del Guest in chiamata di sistema dell'Host, mantenendo la virtualizzazione del solo microcodice ove possibile, ed eliminando lo strato di sistema operativo Guest che pertanto non avrebbe ragione di esistere e velocizzando così l'esecuzione. Questo componente è appunto il **Container Engine.** Da qui la nascita di *Docker* e del concetto di Container. Sul sistema Host anziché girare una o più macchine virtuali Guest con le copie dei rispettivi sistemi operativi, girano dei container che sono appunto degli applicativi le cui chiamate di sistema vengono immediatamente convertite nelle chiamate di sistema dell'Host, che diventa l'unico sistema operativo condiviso. Definirei quindi *Docker* un Container Manager, che nella pratica è implementato come programma che gira sotto Linux, esattamente un demone di Linux che è il **Container Engine.** L'isolamento fra i Containers e la gestione delle risorse condivise è ben gestita da *Docker*

mediante le caratteristiche di Linux. Pertanto su sistema Linux per lanciare Docker



basta a il Docker demon che necessita del Linux Kernel, mentre ovviamente su Windows ritorna il concetto di macchina virtuale, nel senso che poiché il Docker demon è scritto per Linux e gira solo su Linux, bisogna lanciare appunto una macchina virtuale Linux sulla quale gira il demone Docker. Tutto questo viene fatto ovviamente dall'installer di Docker per Windows, che nasconde opportunamente la Linux VM che funziona mediante Hyper-V che è appunto un **hypervisor** nativo di Windows . Sia da Linux(direttamente), che da Windows(via virtualizzazione), si accede a Docker mediante delle **REST API Docker** che la macchina Linux, su cui gira il demone Docker, espone per comunicare col demone stesso. Per interagire con Docker l'applicativo fornisce una interfaccia a riga di comanda, **CLI**, che permette di richiamare le API Docker e di

gestire il Docker demone mediante un linguaggio di scripting o commandi propri della **CLI.** La possibilità di richiamare le API REST di Docker implica che lo strato software contenente queste API funga da server per il Docker che può essere manipolato da qualsiasi client mediante CLI. Circa la virtualizzazione della Linux VM sulla quale gira Docker su sistemi Windows occorre sottolineare che Docker attualmente gira *nativamente* solo su Windows 10 Professional.

La utilità di Docker si apprezza maggiormente quando, ad esempio, si ha necessità di simulare sulla stessa macchina un intero progetto in **Devops** mediante **Continuos Delivery**: sull'Host quindi dovranno risiedere contemporaneamente il server Jenkins, il server Nexus o Artifactory, il server su cui gira un repository GitHub, il server su cui gira l'applicazione, il server su cui gira il database, il server su cui gira l'applicativo, etc., e per ragioni di semplicità e portabilità quindi Docker può essere anche molto usato nella didattica e nell'apprendimento personale delle nuove **tecnologie e metodologie software.**

Queste moderne metodologie di sviluppo software presuppongono la conoscenze delle annesse nuove tecnologie quali appunto Jenkins o Docker e molte altre,, le quali per essere opportunamente utilizzate presuppongono a loro volta inevitabilmente la conoscenza del Sistema Operativo **Linux,** e di come questo gestisce la rete, la memoria, il File System, i privilegi di accesso, la concorrenza dei processi, etc. che sono concetti importantissimi da conoscere nella scrittura dei comandi Linux sia nelle Docker Image che, esempio, nelle pipeline Jenkins.


Windows PowerShell
PS C:\Users\UtenteBusiness> docker --version
Docker version 19.03.5, build 633a0ea
PS C:\Users\UtenteBusiness> docker run hello-world
Hello from Docker!
This message shows that your installation appears to be working correctly.
To generate this message, Docker took the following steps:
1. The Docker client contacted the Docker daemon.
2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
(amd64)
3. The Docker daemon created a new container from that image which runs the
executable that produces the output you are currently reading.
4. The Docker daemon streamed that output to the Docker client, which sent it
to your terminal.
To try something more ambitious, you can run an Ubuntu container with:
$ docker run -it ubuntu bash
Share images, automate workflows, and more with a free Docker ID:
https://hub.docker.com/
For more examples and ideas, visit:
https://docs.docker.com/get-started/
PS C:\Users\UtenteBusiness>
Scrivi qui per eseguire la ricerca
10:48
25/03/2020

Docker in pratica

L'esempio che vado a illustrare serve a familiarizzare con *Docker* ed a fissarne i concetti basilari. Supponiamo di avere una macchina Windows 10 e di installarci *Docker*. L'installazione è relativamente semplice seguendo le indicazioni del sito ufficiale. Una volta installato *Docker* sulla propria macchina Windows 10 si avrà, in sostanza, una macchina virtuale(implementata quasi tutta nativamente da Windows) Linux sulla quale gira appunto il demone *Docker*, che diventa quindi il ***Docker*** **Server** al quale si accede mediante CLI per chiamarne le API. Una volta installato *Docker* la CLI è messa a disposizione da Windows 10 sia via Dos Command che dalla Windows PowerShell(guardare figura precedente) mediante le quali si possono eseguire i comandi CLI per gestire *Docker*. Dopo aver installato *Docker* per verificare che sia correttamente installato(ma non che il demone sia in running) basta eseguire il seguente comando da cmd o da PowerShell:

```
docker --version
```

A seguito dell'installazione per verificare che *Docker* sia in running basta digitare il comando:

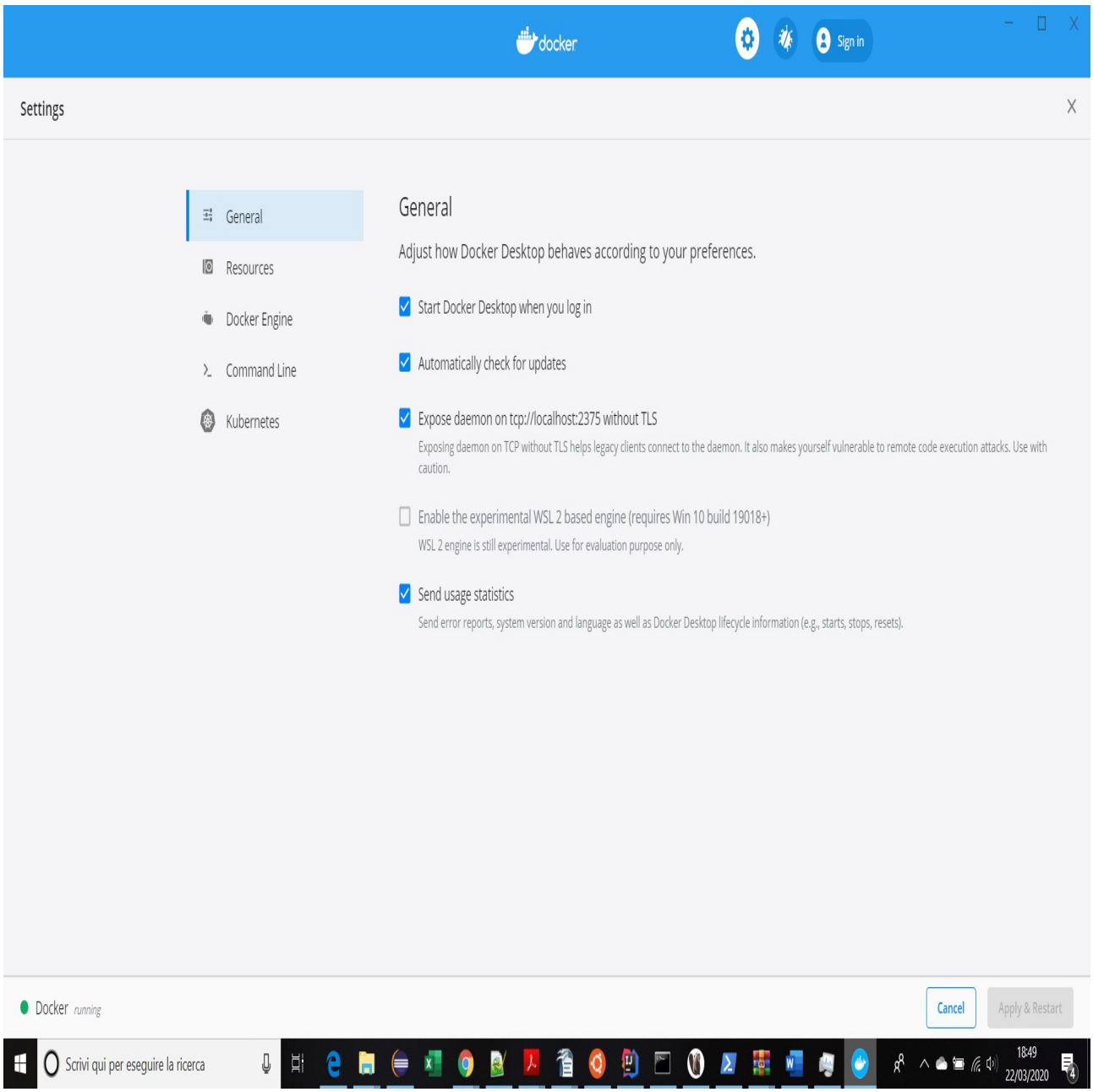
```
docker run hello-world
```

per ragioni di completezza espositiva vorrei connettermi al *Docker* Server(attualmente in running presso la macchina Host Windows 10 di esempio) mediante il sottosistema Ubuntu Linux messo a disposizione dalla store di Windows. Ubuntu Linux for Windows si basa su **Windows Subsystem for Linux** che è una VM Linux ibrida scritta da Microsoft, che rappresenta il kernel Linux. Ibrida perché scritta quasi tutta nativamente ed usando solo un sottinsieme di Hyper-V(quindi in parte virtualizzata), per permettere l' esecuzione di Linux su Windows. Ed è **la stessa struttura usata** da *Docker* per girare su Windows a cui accennavo prima. Una volta installato Ubuntu Linux sulla macchina Windows

10 sulla quale gira Docker, accedo alla shell Ubuntu e devo verificare se Docker è installato sul sottosistema Linux Ubuntu, anche se in questo caso ci servirà solo la CLI(e non lanciare Docker che è già in running sulla macchina Host Windows 10). Ancora una volta dalla shell Ubuntu Linux digito

```
docker --version
```

come si vede nella figura successiva Ubuntu Linux ha incluso Docker e, in questo caso, una versione più aggiornata rispetto a quella dell'Host Windows. Per connettere la VM Ubuntu Linux al Docker Server della macchina Windows 10, e per accedere al Docker Server da Ubuntu mediante CLI, per prima cosa bisogna verificare se il demone viene esposto da Windows 10 stesso, ed a quale indirizzo:porta come si vede in figura.

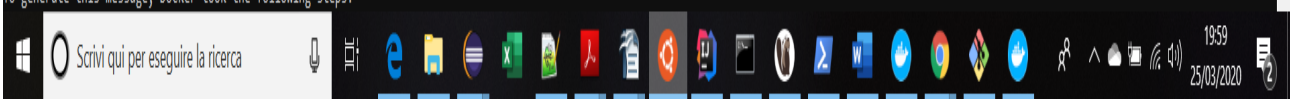


A questo punto bisogna settare(come si vede in figura) la variabile d'ambiente DOCKER_HOST di

Ubuntu via pipe ed aggiornare i comandi Linux mediante il comando source. Per far questo dalla shell di Ubuntu eseguire questi due comandi:

1) echo "export DOCKER_HOST='tcp://0.0.0.0:2375'" >> ~/.bashrc
2)source ~/.bashrc

```
adefazio@DESKTOP-9FE8RCF:~$ docker --version
Docker version 19.03.8, build afacb8b7f0
adefazio@DESKTOP-9FE8RCF:~$ docker run hello-world
docker: Cannot connect to the Docker daemon at unix:///var/run/docker.sock. Is the docker daemon running?.
See 'docker run --help'.
adefazio@DESKTOP-9FE8RCF:~$ echo "export DOCKER_HOST='tcp://0.0.0.0:2375'" >> ~/.bashrc
adefazio@DESKTOP-9FE8RCF:~$ source ~/.bashrc
adefazio@DESKTOP-9FE8RCF:~$ docker run hello-world
Unable to find image 'hello-world:latest' locally
latest: Pulling from library/hello-world
1b930d010525: Pull complete
Digest: sha256:f9dfddf63636d84ef479d645ab5885156ae030f611a56f3a7ac7f2fdd86d7e4e
Status: Downloaded newer image for hello-world:latest

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
 1. The Docker client contacted the Docker daemon.
 2. The Docker daemon pulled the "hello-world" image from the Docker Hub.
    (amd64)
 3. The Docker daemon created a new container from that image which runs the
    executable that produces the output you are currently reading.
 4. The Docker daemon streamed that output to the Docker client, which sent it
    to your terminal.

To try something more ambitious, you can run an Ubuntu container with:
 $ docker run -it ubuntu bash

Share images, automate workflows, and more with a free Docker ID:
 https://hub.docker.com/

For more examples and ideas, visit:
 https://docs.docker.com/get-started/

adefazio@DESKTOP-9FE8RCF:~$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
adefazio@DESKTOP-9FE8RCF:~$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
adefazio@DESKTOP-9FE8RCF:~$ docker ps -a
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS                      PORTS               NAMES
3e5ced9ae44c        hello-world         "/hello"            2 hours ago         Exited (0) About an hour ago                    wonderful_ellis
d6d2925093e1        hello-world         "/hello"            2 hours ago         Exited (0) About an hour ago                    condescending_herschel
e7380fa8cf1c        hello-world         "/hello"            2 hours ago         Exited (0) 2 hours ago                          crazy_mclean
adefazio@DESKTOP-9FE8RCF:~$ docker run hello-world

Hello from Docker!
This message shows that your installation appears to be working correctly.

To generate this message, Docker took the following steps:
```



Il client Docker è opportunamente configurato in Ubuntu. Basta lanciare il comando Docker

```
docker run hello-world
```

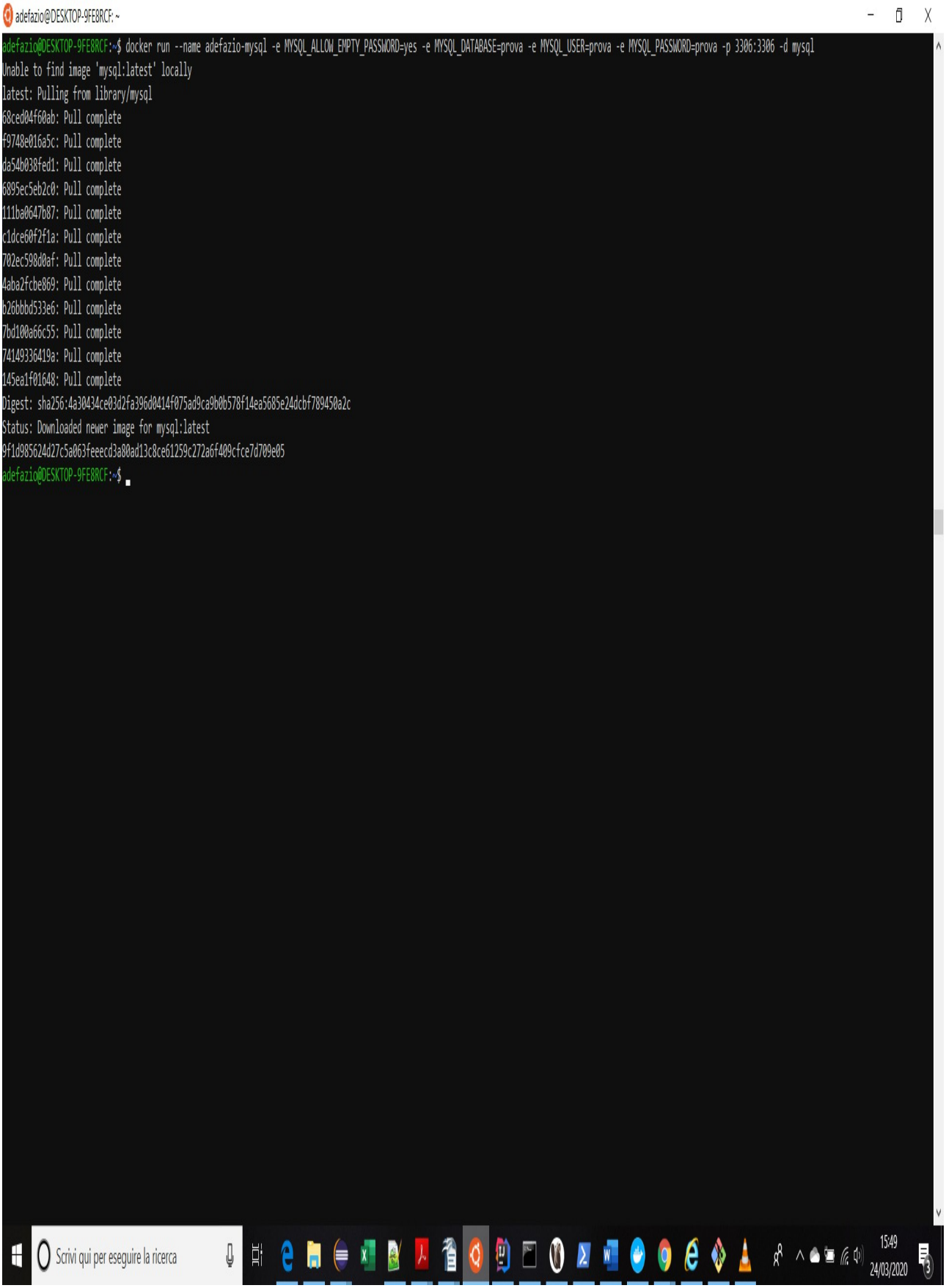
e si vedrà che il sistema è stato opportunamente configurato. Sottolineo ovviamente che una Docker CLI, a seguito dell'installazione in Windows, è già funzionante sia nella PowerShell che nel Windows Command Prompt o CMD: configurarla anche in Ubuntu serve solo a scopi di migliore comprensione dell' ecosistema Docker.

A questo punto qualche aspetto teorico fondamentale alla comprensione del seguito.

Come spiegato sopra Docker permette di simulare, e virtualizzare, sistemi Operativi, databases, linguaggi di programmazione, etc. L'insieme di files di cui necessita Docker per istanziare e gestire un Container è appunto l'image. Per fare un paragone con Java e gli oggetti, definirei una Docker image il sorgente Java, sorgente che viene compilato con dei parametri mediante una Docker build, e l' oggetto viene istanziato con i parametri mediante il comando Docker run.

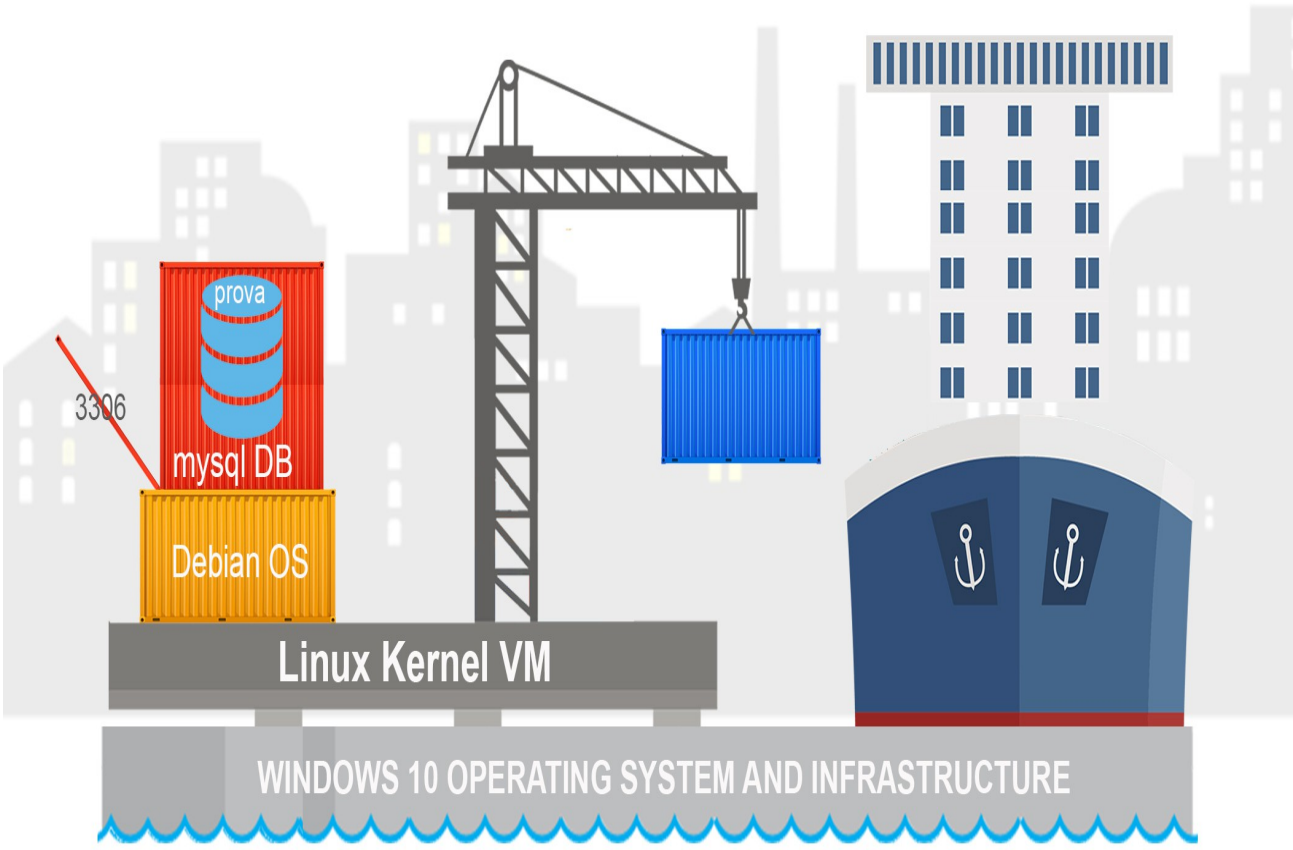
Come primo esempio vorrei istanziare nel contesto Docker, via CLI, una macchina(completa di SO) che simuli un database MySQL, quindi un MYSQL Container, per poi accedervi mediante database client dalla macchina Windows Host su cui già gira Docker come abbiamo visto. Pertanto ritorno sulla Shell di Ubuntu dove abbiamo preconfigurato la CLI Docker ed eseguo il comando:

```
docker run --name adefazio-mysql -e MYSQL_ALLOW_EMPTY_PASSWORD=yes -e MYSQL_DATABASE=prova -e MYSQL_USER=prova -e MYSQL_PASSWORD=prova -p 3306:3306 -d mysql
```

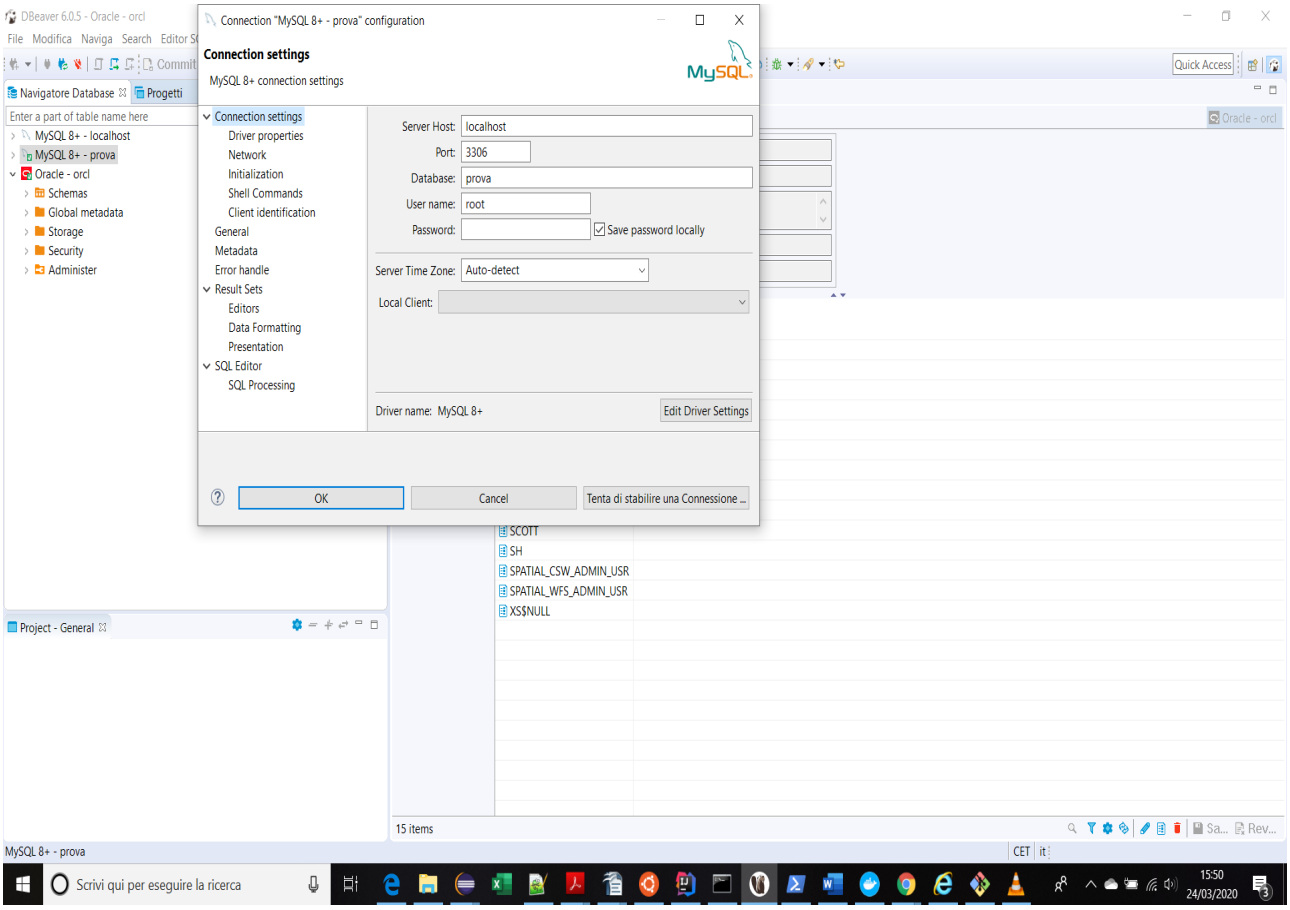


Questo comando molto potente fa contemporaneamente tante cose: verifica se l'immagine mysql(il

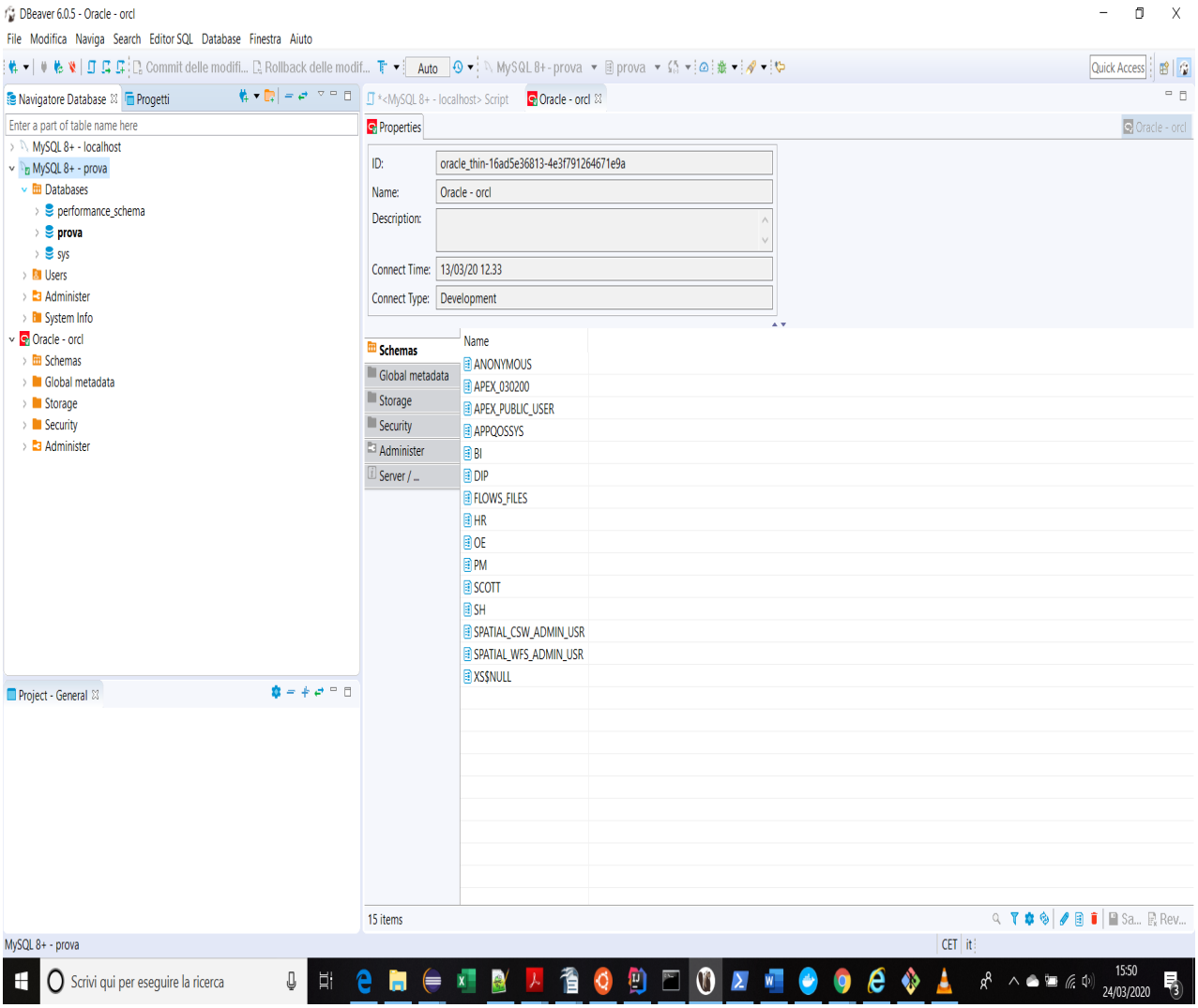
parametro successivo a -d) è già stata scaricata ed installata(pull), qualora non lo sia(come in questo caso) si connette al Docker Hub, la scarica, la installa, crea il nuovo container e lo lancia! Il Docker Hub è un registro pubblico di immagini Docker pubblicamente disponibili simile, per



fare un paragone, al maven repository remoto. Come Maven anche *Docker* verifica ovviamente che l'immagine sia presente prima nel registro locale, e cioè verifica che ne sia già stata fatta la pull. A parte gli altri parametri che si autoesplicano, quello più importante è quello che segue -p responsabile del **binding** fra le porte della macchina Host(in questo caso Windows 10), ed il container. La porta 3306 del Container MYSQL è l'entrypoint del container MYSQL, cioè la porta che il container MYSQL espone, e viene **linkata** alla porta 3306 del Windows 10 Host. Sottolineo che questo container MYSQL è una simulazione di una intera macchina su cui gira il sistema operativo **Linux Debian** sul quale a suo volta gira MYSQL database, ogni Container/Image rappresentante qualsiasi componente architetturale(un database, un JVM, un'applicazione, un server, etc.) è ovviamente implementato su un OS Linux simulando in tal modo una macchina indipendente a tutti gli affetti.
Fra la miriade di client utilizzabili per verificare che il container MYSQL sia in running ed accessibile uso personalmente Dbeaver con questi parametri di connessione.



Come da noi richiesto viene creato un database il cui nome è prova con le credenziali espresse nei



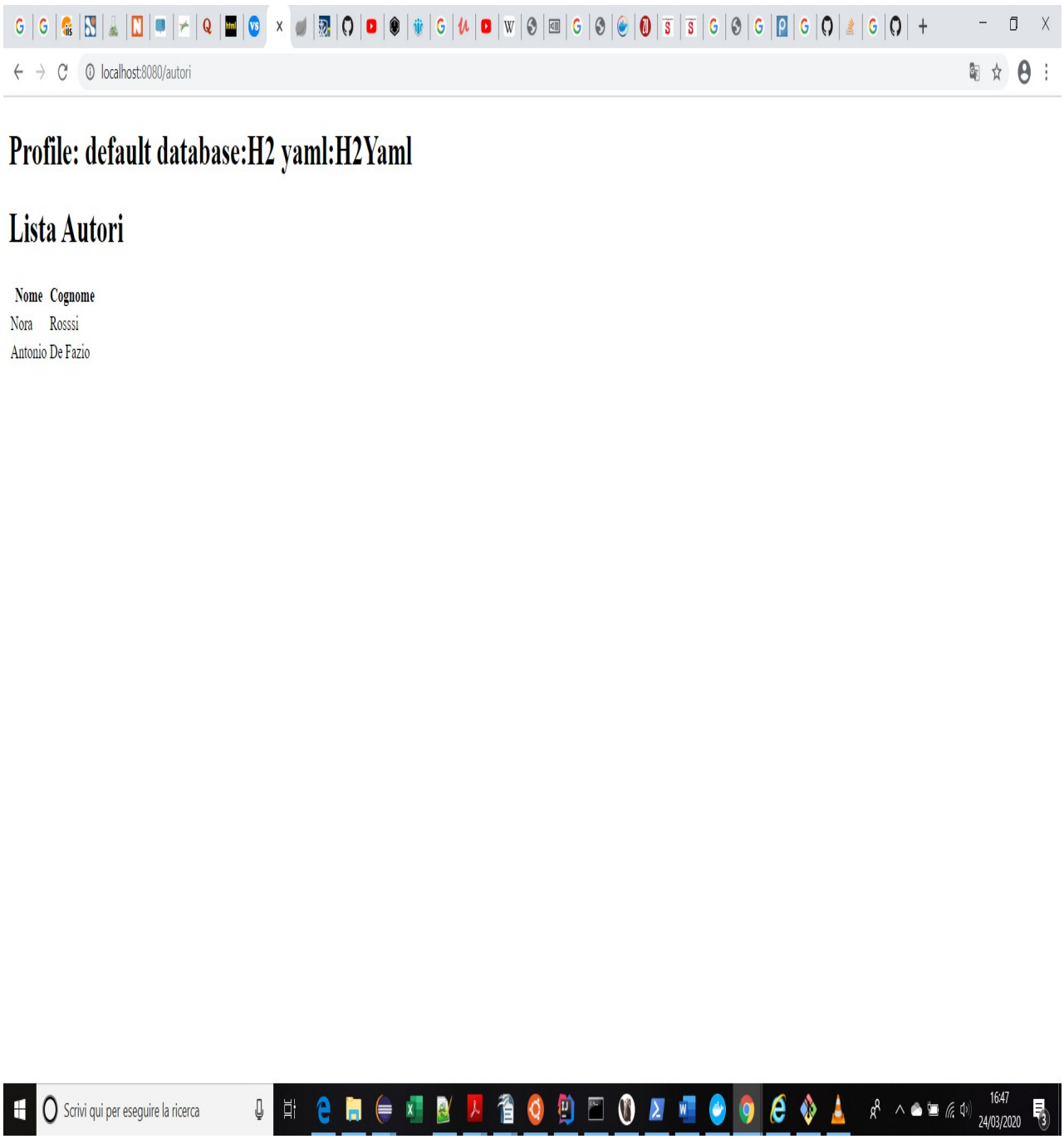
parametri del *Docker* run, il root ha password null perchè è solo un test.

Ora vorrei sfruttare un semplice progetto SpringBoot e vedere come il framework si relaziona al mondo *Docker*. Progetto disponibile al link:

https://github.com/antoniodefazio/adefazioworkspace

Questo è un progetto Maven creato col framework Spring Boot pertanto l'artefatto è un figlio di `spring-boot-starter-parent`.L'applicativo è una web application che mostra banalmente a video la lista di Autori, o la lista di Libri, persistiti su db. Vi sono essenzialmente 3 profili Spring: 2 configurati nel file Yaml, mysql, e docker, ed ovviamente quello di default che configuro nel file di properties per ragioni didattiche. I vari profili discriminano qualche bean e le differenti connessioni a db, per ragioni

didattiche, ma a livello ingegneristico, come è noto, gli indirizzi del db è bene che stiano nelle configurazioni del server. La tecnologia usata fronted è Thymeleaf, mentre i links al database sono ovviamente funzione del profilo in esecuzione. Dopo averlo scompattato ed importato in qualsiasi IDE come Maven project, e dopo aver fatto la maven install, vi sono tanti modi per lanciarlo come spiego nel precedente tutorial. La classe SvilBoot è un Component Spring che inizializza i valori del database. Come primo esempio lanciamo il profilo default che implica l'uso del database H2 che è *embedded* in Spring Boot.

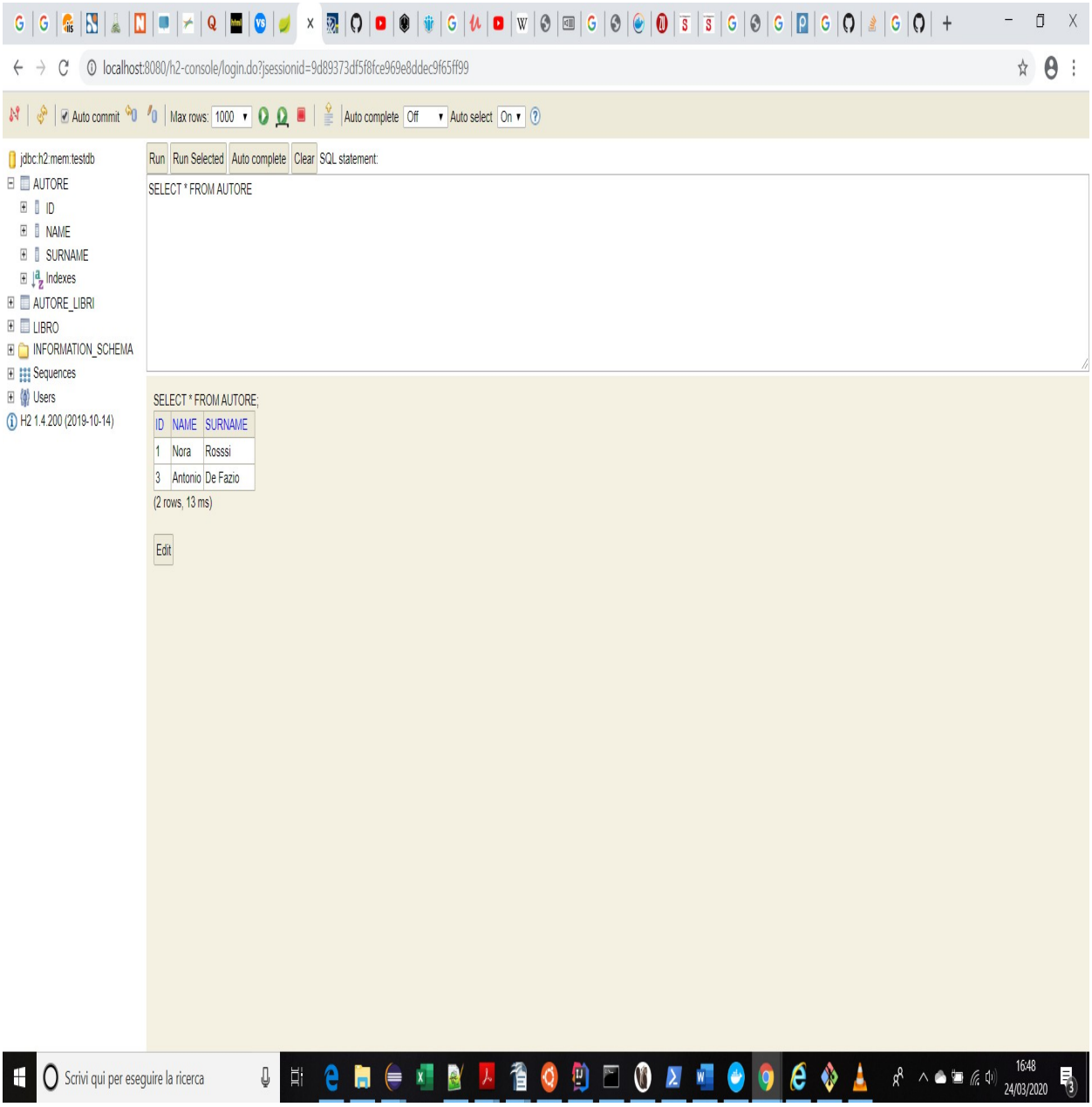


Personalmente come test uso l'ultima versione di Eclipse per progetti JEE che ha esattamente il tasto Rus as->Spring Boot App e faccio lo switch fra i diversi profili settando la variabile:

spring.profiles.active nel file di properties. Lancio il profilo default ed accedo agli indirizzi:
http://localhost:8080/libri
http://localhost:8080/autori

Nella figura precedente vi è il frontend.

Questo è il database H2:



A questo punto basta lanciare il profilo mysql e si potrà notare come il progetto Web si connetta al MYSQL locale che è appunto il *Docker* Container. Lascio questo al lettore come esercizio. Pertanto si può apprezzare come con una banale riga di codice si possa creare e lanciare un database MYSQL grazie a *Docker*, per poi usarlo per la nostra Web app. Questo per la portabilità e il testing della nostra applicazione è fondamentale!

Proviamo a fare qualcosa di più affascinante con *Docker*: ***iniettare*** la nostra applicazione in un container *Docker*. Per prima cosa vorrei simulare una situazione reale, e cioè una Web app che risiede su una macchina sulla quale gira il sistema operativo Linux. Pertanto lancio, nel contesto

Docker, il container di un sistema operativo Linux concepito per usi professionali:Centos. Mediante il comando:

docker run -d centos tail -f /dev/null

il tail finale è un comando linux che serve per non fermare il Centos subito dopo il run. Allora abbiamo lanciato il Container Centos.

Ora viene il bello per capire la potenza e la semplicità di Docker!. Attualmente quindi nel contesto Docker abbiamo in running un container di database MYSQL, ed un container di OS Linux Centos. Per prima cosa vediamo come installare e lanciare la nostra Web app sul Container Linux Centos con profilo di default e database embedde H2.

La nostra web app è, grazie a Spring Boot, un *fat* .jar con un Servlet Container Tomcat *embedded* pertanto non vi è praticamente bisogno di un server su cui fare il deploy. Copio il file provaDocker.jar in una qualche cartella della VM Linux Ubuntu, **sto usando questa come CLI ma si può fare la stessa cosa anche da Dos Command e usare la cmd come CLI o dalla PowerShell di Windows.** Nella stessa cartella dove ho copiato la nostra applicazione provaDocker.jar edito e salvo un file che chiamo "Dockerfile"(con la D maiuscola) in cui vi sono una serie di script Docker, questo file equivale a creare una image *custom* che serve a generare un Container *custom*. Il file è composto da queste sole 5 righe:

```
FROM centos
RUN yum install -y java
VOLUME /tmp

ADD provaDocker.jar myapp.jar

RUN sh -c 'touch /myapp.jar'
```

ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-Dspring.profiles.active=default","-jar","/myapp.jar"]

La clausula FROM va a stabilire da quale immagine si deve partire, e cioè su quale immagine deve collocarsi l' immagine in questione(quindi da quale container), simile al concetto di ereditarietà in Java. In questo caso sarà *figlia* del Centos creando un altro Container Centos e la cui **immagine verrà condivisa con quella del Centos in running**, Docker funziona ad immagini che estendono altre ed è come se fossero immagini immutabili in quanto non è concesso cambiare nulla delle immagini a livello superiore, ma sostanzialmente possiamo modificare appunto solo l'ultima classe nella gerarchia, quindi il file Docker. Il primo comando della image mysql è infatti FROM debian:buster-slim. che è appunto il sistema operativo che il Team MYSQL ha scelto di usare come sottostante il container del database MYSQL. Successivamente vi è il CLI command RUN che abbiamo visto e che scarica e installa la JDK. La VOLUME crea uno spazio nel file system per i metadati di questa image. La direttiva ADD serve appunto ad aggiungere il jar, che attualmente è nel client Docker, all'interno del Container corrente. Il quarto comando serve per lanciare una shell command ed impostare data e ora al nuovo file myapp.jar che è la copia del provaDocker.jar nel Container. L'ultimo comando, ENTRYPOINT, serve appunto a stabilire il comando da lanciare in fase di startup del Container, cioè la nostra applicazione Java, con gli opportuni parametri. La variabile java.security.egd serve solo a lanciare più velocemente il Tomcat. Sottolineo che il Centos Container di questa immagine crea un'atra istanza del Centos e condivide in lettura il Sistema Operativo Centos Container già in running, qualora non si avesse questo Container Centos in running, questa image **lancerebbe comunque Centos Container**.

Poi bisogna fare la build del file e dare un nome(un tag) all'image, che chiameremo spring-boot-

docker, col comando:

```
docker build -t spring-boot-docker .
```

Questo costruisce questa image e la salva nel registro Docker locale, dandole un nome, o meglio un tag(-t), Il file di costruzione della image la Docker CLI lo trova andando a cercare un file, nella directory corrente(.), il cui nome è "Dockerfile" con la D maiuscola. Poi si lancia il comando:

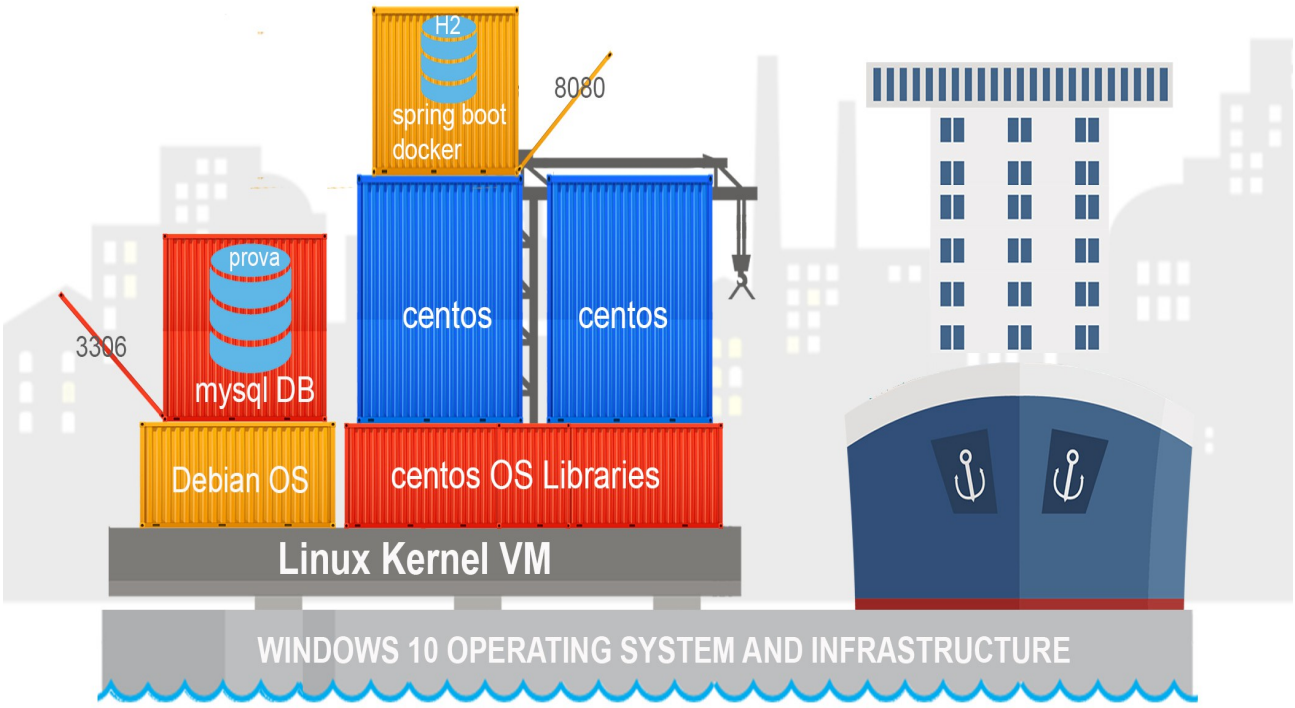
```
docker run -d -p 8080:8080 spring-boot-docker
```

che sostanzialmente avvia la nostra applicazione Spring Boot all'interno del Container Linux Centos, pur essendo un Container a sé stante.

Questo comando lancia il terzo container in Docker: spring-boot-docker che in sostanza rappresenta la nostra web app(jdk+provaDocker.jar su Centos Container) e condivide l'immagine col Container Centos **senza duplicarlo.**

Ora ci connettiamo all'applicazione dall' Host Windows agli indirizzi:
http://localhost:8080/libri
http://localhost:8080/autori

Se avessimo lanciato l'applicazione Spring Boot con profilo mysql, cambiando i parametri nell' ENTRYPOINT, non funzionerebbe in quanto l'applicazione gira sul Container Linux Centos che **non ha al suo interno** un database MYSQL in running. Abbiamo comunque già un container MYSQL in running e potremmo utilizzarlo ma, essendo esterno alla macchina su cui gira l'applicazione, non si può usare l'indirizzo localhost come nel profilo mysql, **e bisogna fare in modo di richiamare dal container Centos il container MYSQL,** esterno a Centos, ma che sta nello stesso ecosistema Docker.



Per fare questo si deve **stoppare** il solo container spring-boot-docker(cioè la web app) ed **eliminare** il *Docker*File in quanto bisogna crearne un altro, per cambiare la ENTRYPOINT e cambiare il profilo in quanto l'url del db è differente. Poi bisogna ovviamente ripetere il processo di build e run. Il terzo profilo è **docker**, che ha un Url differente per il db MYSQL(guardare il file Yaml del progetto). Il *Dockerfile* è quindi:

```
FROM centos
RUN yum install -y java
VOLUME /tmp

ADD provaDocker.jar myapp.jar

RUN sh -c 'touch /myapp.jar'
```

ENTRYPOINT ["java","-Djava.security.egd=file:/dev/./urandom","-

Dspring.profiles.active=docker","-jar","/myapp.jar"]

Mentre il run cambia:

docker run -p 8080:8080 --name mysqllink --link adefazio-mysql:mysql -d spring-boot-docker

L'opzione **link** del *Docker* run serve appunto a linkare il container mysql con spring-boot-docker

mediante appunto adefazio-mysql che diviene, lato spring-boot-docker, il nome della macchina su

cui gira mysql, che invece di essere una macchina reale è appunto il Container MYSQL.

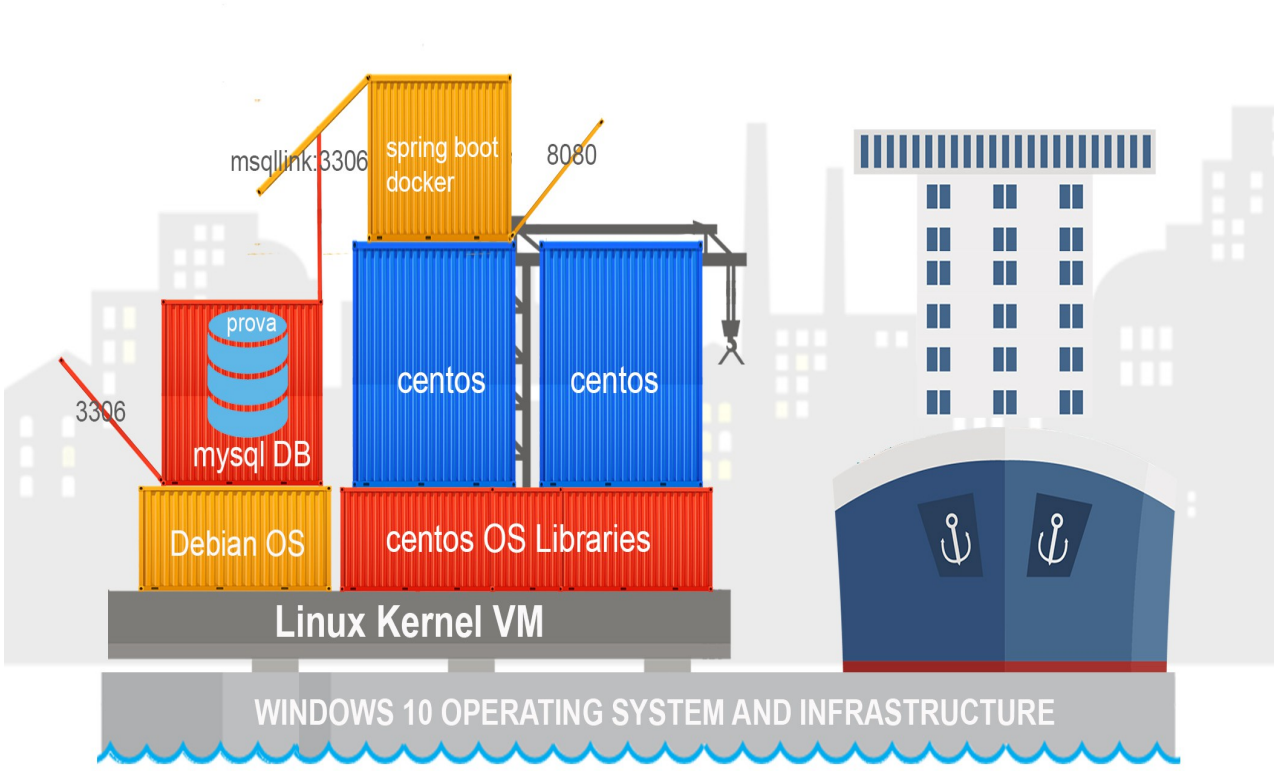
Sostanzialmente Docker mantiene una sorta di ***DNS interno*** ad ogni Container nel quale mappa i nomi delle altre macchine(Containers quindi) per le connessioni. Nel file Yaml di configurazione dei profili si vede che l'url al database del profilo **docker** è appunto:

jdbc:mysql://adefazio-mysql:3306/prova

Ora ci connettiamo all'applicazione dall' Host Windows agli indirizzi:
http://localhost:8080/libri
http://localhost:8080/autori

Inoltre ci si può connettere col db client al MYSQL Container, editare il database e vedere come l'applicativo punti realmente al Docker Container MYSQL. Pertanto dal sistema Host Windows richiamiamo un Container custom di nome spring-boot-docker(cioè Linux Centos +java + provaDocker.jar) che a sua volta carica dati da un database esterno al Container, quindi lato Centos il database è remoto, che è un altro Container MYSQL. Con estrema semplicità, mediante Docker, si sta simulando, **su una macchina** Windows 10 che funge da Host, contemporaneamente:

1)una macchina Linux Centos, quindi quello che si definisce un *blank operating system*;
2)una macchina Linux Centos con su una applicazione java;



3)una macchina Debian Os con su un database;

per poi connettere fra di loro gli ultimi due: si pensi all'enorme spreco di risorse e tempo che sarebbe derivato dal fare la stessa cosa con le macchine virtuali. Tutta l'architettura inoltre è estremamente portabile, bastano solo il jar e qualche istruzione *Docker*.

Buon lavoro.